

# Tutoriel pour apprendre à développer les services REST avec Spring Boot et Spring RestTemplate

Projet Spring Boot



Par Nguimgo Bertrand 

Date de publication : 7 mars 2018

TOUT PUBLIC

Le but de cet article est de présenter comment développer les services REST à partir de Spring Boot et de Spring RestTemplate. Le client et le serveur seront développés dans deux applications distinctes afin de montrer une séparation claire qui existe entre les deux parties.

Un espace de dialogue est disponible pour apporter votre contribution ici: [Commentez](#)

I - Première partie : Le serveur.....	3
I-A - Introduction.....	3
I-A-1 - Cahier des charges.....	3
I-A-2 - Technologies utilisées.....	4
I-A-3 - Pourquoi utiliser Spring Boot ?.....	4
I-A-4 - Présentation des services REST.....	5
I-B - Création de l'application.....	5
I-B-1 - Création de l'architecture.....	5
I-B-1-a - Configuration et exécution.....	7
I-B-1-b - Structure de l'application.....	9
I-B-1-c - Création du service.....	11
I-B-1-d - Test de la configuration.....	11
I-B-2 - Création des couches.....	12
I-B-2-a - Création du model.....	12
I-B-2-b - Création de la base de données.....	17
I-B-2-c - Création de la couche DAO.....	18
I-B-2-d - Couche de services.....	18
I-B-3 - Mise en place du service REST.....	21
I-B-3-a - Création du filtre Cross Domain.....	21
I-B-3-b - Nouveautés dans les requêtes HTTP avec Spring.....	24
I-B-3-c - Création d'un contrôleur par défaut.....	24
I-B-3-d - Service d'extraction de tous les utilisateurs.....	25
I-B-3-e - Service création d'un utilisateur.....	26
I-B-3-f - Recherche d'un utilisateur par son login.....	28
I-B-3-g - Service modification d'un utilisateur.....	29
I-B-3-h - Suppression d'un utilisateur.....	29
I-C - Gestion des exceptions.....	30
I-C-1 - Pourquoi utiliser @ControllerAdvice.....	30
I-C-2 - Test des services avec exceptions.....	34
I-D - Tests unitaires des couches.....	36
I-D-1 - Tests unitaires couche DAO.....	37
I-D-2 - Tests unitaires couche service.....	39
I-D-3 - Tests unitaires du contrôleur.....	42
I-D-4 - Résumé sur les tests unitaires.....	47
I-E - Tests d'intégration services REST.....	47
II - Consommation des services.....	52
II-A - Introduction.....	52
II-B - Création du projet client.....	52
II-C - Création du contrôleur.....	56
II-C-1 - Contrôleur par défaut.....	56
II-C-2 - Contrôleur UserController.....	59
II-C-3 - Configuration de l'application.....	61
II-C-4 - Création des pages web.....	62
II-D - Test de l'application.....	62
II-D-1 - Serveur démarré.....	63
II-D-2 - Serveur arrêté.....	63
II-D-3 - Création d'un utilisateur.....	63
II-D-4 - Test de connexion.....	64
II-D-5 - Page de connexion réussie.....	64
III - Conclusion.....	65
III-A - Sources et références.....	65
III-B - Remerciements.....	65

## I - Première partie : Le serveur

Il est question dans cette partie de développer un service REST, et ensuite de le consommer dans la seconde partie de ce tutoriel.

### I-A - Introduction

La plupart des applications d'entreprises sont des applications client/serveur. Pour développer ces applications, les deux solutions les plus dominantes sur le marché sont les web services **REST (Representational State Transfer)** et web services SOAP (**Simple Object Access Protocol**). SOAP est un protocole très normé et plus ancien que REST. L'objectif de cet article n'est pas de faire une étude comparée des deux technologies, mais avant de passer des heures à choisir entre SOAP et REST, il faut prendre en compte le fait que certains services web supportent l'un et pas l'autre. Par exemple, SOAP fonctionne en environnement distribué (Point à multipoint) alors que REST nécessite une communication directe point à point, basée sur l'utilisation des URI. De plus en plus d'entreprises préfèrent les services REST pour leur simplicité de mise en œuvre, et c'est l'une des raisons pour lesquelles l'idée m'est venue de rédiger cet article et surtout de montrer comment mettre en œuvre un service REST. Pour plus d'informations sur la technologie SOAP et son implémentation, je vous conseille d'excellents tutoriels de [Mickael Baron](#).

À la fin de cet article, vous devrez être capable de développer des web services REST en utilisant **Spring Boot**, savoir **automatiser les tests unitaires et les tests d'intégration**, **être capable de mieux gérer les exceptions** dans une application REST et le tout en **Java8**.

Pour ceux qui ont déjà les connaissances avancées sur les services REST, vous pouvez passer [directement à la pratique](#)

Toutes les sources (client et serveur) sont [disponibles en téléchargement](#)

### I-A-1 - Cahier des charges

Il s'agit de développer un portail web d'inscription et de connexion en utilisant les **services RESTful** côté serveur et **Spring RestTemplate** côté client.

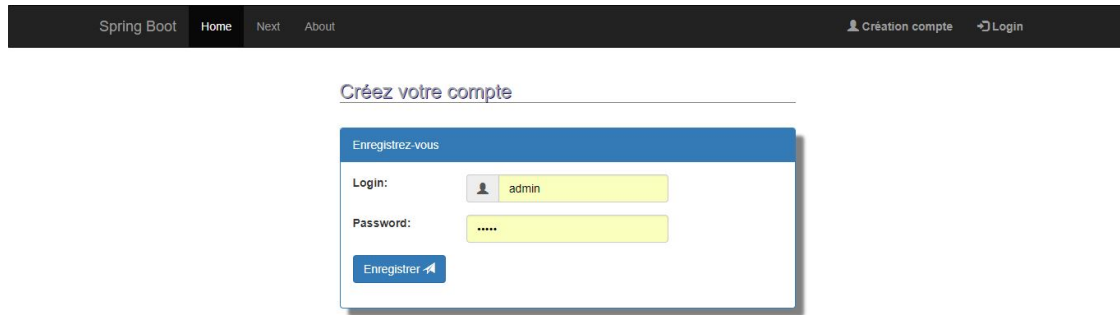


*Le serveur sera développé dans une application à part et le client dans une autre. Je vais ensuite mettre en relation les deux services grâce à l'URL du serveur et le Framework Spring RestTemplate. Ce choix d'architecture permet de bien mettre en évidence la séparation des services REST et leur consommation par les clients.*



*Dans cette partie, il n'y aura aucun fichier JSP (Java Server Page) à écrire. Ça se fera côté client. Mais, je peux déjà vous montrer à quoi va ressembler l'écran d'accueil de l'application à travers l'image ci-dessous :*

Voici à quoi ressemblera la page de connexion.



*Écran de création du compte et de connexion*

## I-A-2 - Technologies utilisées

### Technologies à implémenter dans ce projet

- Spring Boot-1.5.9-RELEASE.
- Java-1.8.
- Frameworks : Maven, SpringMVC, Spring-RestTemplate, Mockito, JSON, Boomerang.
- Base de données embarquée, H2 et hsqldb.
- Tomcat-8.
- IDE : Eclipse-Mars-4.5.0.
- La plupart des technologies sont embarquées par Spring Boot.

## I-A-3 - Pourquoi utiliser Spring Boot ?

**Définition** : **Spring** est un Framework de développement d'applications Java, qui apporte plusieurs fonctionnalités comme Spring Security, SpringMVC, Spring Batch, Spring Ioc, Spring Data, etc. Ces Frameworks ont pour objectif de faciliter la tâche aux développeurs. Malheureusement, leurs mises en œuvre deviennent très complexes à travers les fichiers de configuration XML qui ne cessent de grossir, et une gestion des dépendances fastidieuse. C'est pour répondre à cette inquiétude que le projet Spring Boot a vu le jour.

**Définition** : **Spring Boot** est un sous projet de Spring qui vise à rendre Spring plus facile d'utilisation en élimant plusieurs étapes de configuration. L'objectif de Spring Boot est de permettre aux développeurs de se concentrer sur des tâches techniques et non des tâches de configurations, de déploiements, etc. Ce qui a pour conséquences un gain de temps et de productivité (avec Spring Boot, il est très facile de démarrer un projet n-tiers).

### Spring Boot apporte à Spring une très grande simplicité d'utilisation :

- 1 Il facilite notamment la création, la configuration et le déploiement d'une application complète. **On n'a plus besoin des fichiers XML à configurer** (pas besoin du fichier du descripteur de déploiement web.xml dans le cas d'une application web). Nous verrons plus bas comment cela est possible.
- 2 Spring Boot permet de **déployer très facilement une application dans plusieurs environnements sans avoir à écrire des scripts**. Pour ce faire, une simple indication de l'environnement (développement ou production) dans le fichier de propriétés (.properties) suffit à déployer l'application dans l'un ou l'autre environnement. Ceci est rendu possible grâce à la notion de profil à déclarer toujours dans le fichier de propriétés. Je vous présenterai des exemples de cas d'utilisation.
- 3 Spring Boot possède **un serveur d'application Tomcat embarqué** afin de faciliter le déploiement d'une application web. Il est possible d'utiliser un serveur autre ou externe, grâce à une simple déclaration dans le fichier pom.xml.
- 4 Spring Boot permet de mettre en place **un suivi métrique de l'application** une fois déployée sur le serveur afin de suivre en temps réel l'activité du serveur, ceci grâce à **spring-boot-starter-actuator**.

## I-A-4 - Présentation des services REST

**Les services REST** représentent un style d'architecture pour développer des services web. Une API qui respecte les principes REST est appelée **API-RESTful**.

Les principes clés de REST impliquent la séparation de l'API en ressources logiques. Ce qui revient à penser à comment obtenir chaque ressource.

Une ressource est un objet ou une représentation d'objets contenant éventuellement des données. Exemple : un employé d'une société est une ressource. La manipulation de ces ressources repose sur le protocole HTTP à travers les méthodes d'actions **GET, POST, PUT, PATCH, DELETE**.

Pour obtenir une ressource, il faut déjà l'identifier à travers une URL et bien la nommer.

Une **URL (Uniform Resource Locator)** indique le chemin vers une ressource. Cette ressource n'est pas toujours disponible. Lorsque l'URL pointe vers une ressource disponible, on parle d'une **URI**.

Une **URI (Uniform Resource Identifier)** est l'identifiant unique de ressource sur un réseau (**URI = URL + Ressource**). Une URI est donc une URL qui pointe vers une ressource bien identifiée.

Dans l'implémentation des services REST, une bonne identification de la ressource est l'utilisation des noms et non des verbes. Par exemple, pour notre portail de connexion, j'aurai besoin d'une ressource utilisateur nommée User et d'une URL qui pointe vers cette ressource (l'ensemble formera l'URI de la ressource User), ce qui donne par exemple :

<http://localhost:8080/springboot-restserver/user>

Une fois que la ressource est identifiée, on a besoin de la manipuler. Par exemple la mettre à jour, supprimer, ou en créer une autre.

**Pour ce faire on utilise les méthodes HTTP suivantes :**

- **GET /users** -- extraction de tous les utilisateurs.
- **GET /users/1** -- extraction de l'utilisateur ayant l'identifiant 1.
- **DELETE /users/1** -- suppression de l'utilisateur 1.
- **POST /users** -- création d'un nouvel utilisateur.
- **PUT /users/1** -- mise à jour de l'utilisateur 1.
- **PATCH /users/1** -- mise à jour partielle de l'utilisateur 1.



*Il est conseillé d'utiliser les noms au pluriel (users et non user) pour un bon nommage, qui n'est certes pas obligatoire, mais apporte une certaine lisibilité entre le mapping de l'URL et la ressource.*

## I-B - Création de l'application

Dans cette partie, je vais créer l'architecture de base de l'application à partir Spring Boot.

### I-B-1 - Création de l'architecture

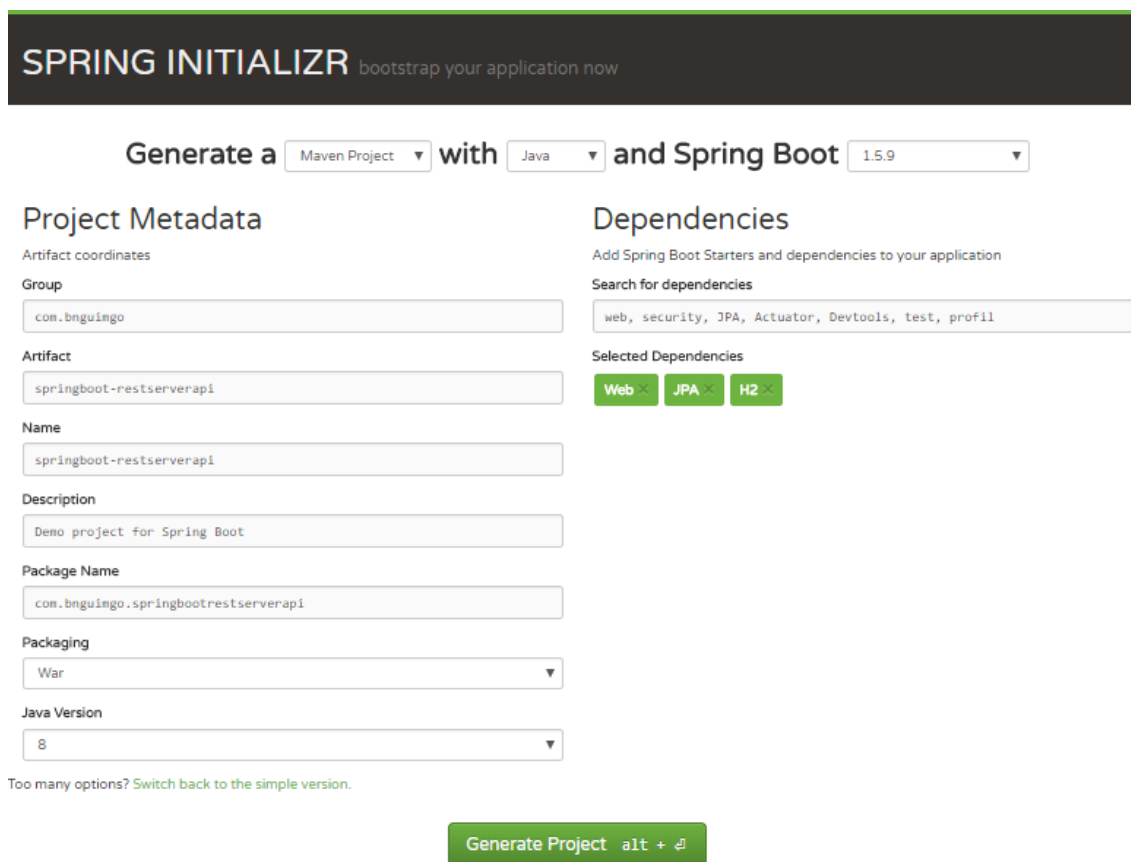
**Pour créer l'application, vous avez plusieurs possibilités:**

- Créer à partir de l'**IDE Eclipse** un projet Maven simple et le compléter avec les dépendances dont vous avez besoin.
- Créer un projet à partir du générateur fourni par Spring Boot à l'adresse : [🇬🇧 Spring Boot Quick Start](#).

C'est cette deuxième option que nous allons utiliser. L'avantage, c'est qu'on peut choisir très aisément toutes les dépendances directement.

**i** Pour voir toutes les dépendances proposées par Spring Boot, dérouler le menu en cliquant sur: **"Switch to the full version"**. Vous pourriez aussi changer de version Spring Boot après génération, car Spring Boot ne propose pas la possibilité de choisir les versions antérieures lors de la génération.

Voici la capture web de génération basique d'un projet web avec Spring Boot :



The screenshot shows the Spring Initializr web interface. At the top, it says "SPRING INITIALIZR bootstrap your application now". Below this, there's a header "Generate a Maven Project with Java and Spring Boot 1.5.9".

The main content is divided into two columns:

- Project Metadata:**
  - Artifact coordinates:
    - Group:
    - Artifact:
    - Name:
    - Description:
    - Package Name:
    - Packaging: War
    - Java Version: 8
- Dependencies:**
  - Add Spring Boot Starters and dependencies to your application
  - Search for dependencies:
  - Selected Dependencies:
 

Web
JPA
H2

At the bottom, there's a green button "Generate Project" with a small icon. Below the button, it says "Too many options? [Switch back to the simple version](#)."

### Génération du squelette projet avec Spring Boot

Comme le montre la capture, il y a trois dépendances principales qui ont été sélectionnées :

- **Web** : pour tout ce qui est projet web (Spring Boot va ramener toutes les dépendances nécessaires pour une application web).
- **JPA** : pour la couche de persistance.
- **H2** : pour la base de données.

J'ai directement choisi un packaging war puisque notre application sera une application web et va tourner sur un serveur d'application Tomcat.

Comme on peut le constater dans la capture, l'application sera développée en Java8.

Pour importer le projet dans Eclipse, décompressez-le et suivez la démarche ci-dessous:

Fichier --> Import --> Maven --> Existing Maven Projects --> Indiquez le répertoire du pom.xml --> Finish

## I-B-1-a - Configuration et exécution

### Présentation du contenu du fichier pom.xml

L'élément central dans un projet Maven est le fichier **pom.xml**. Ci-dessous le fichier pom.xml généré :

#### contenu du fichier pom.xml généré

```

1.
2. <?xml version="1.0" encoding="UTF-8"?>
3. <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance"
4.     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
maven-4.0.0.xsd">
5.     <modelVersion>4.0.0</modelVersion>
6.
7.     <groupId>com.bnquimgo</groupId>
8.     <artifactId>springboot-restserver</artifactId>
9.     <version>0.0.1-SNAPSHOT</version>
10.    <packaging>war</packaging>
11.
12.    <name>springboot-restserver</name>
13.    <description>Demo project for Spring Boot</description>
14.
15.    <parent>
16.        <groupId>org.springframework.boot</groupId>
17.        <artifactId>spring-boot-starter-parent</artifactId>
18.        <version>1.5.9.RELEASE</version>
19.        <relativePath/> <!-- lookup parent from repository -->
20.    </parent>
21.
22.    <properties>
23.        <start-
class>com.bnquimgo.restclient.com.bnquimgo.springbootrestserver.SpringbootRestserverApplication</start-
class><!-- cette déclaration est optionnelle -->
24.        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
25.        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
26.        <java.version>1.8</java.version>
27.    </properties>
28.
29.    <dependencies>
30.        <dependency>
31.            <groupId>org.springframework.boot</groupId>
32.            <artifactId>spring-boot-starter-data-jpa</artifactId>
33.        </dependency>
34.        <dependency>
35.            <groupId>org.springframework.boot</groupId>
36.            <artifactId>spring-boot-starter-web</artifactId>
37.        </dependency>
38.
39.        <dependency>
40.            <groupId>com.h2database</groupId>
41.            <artifactId>h2</artifactId>
42.            <scope>runtime</scope>
43.        </dependency>
44.        <dependency>
45.            <groupId>org.springframework.boot</groupId>
46.            <artifactId>spring-boot-starter-tomcat</artifactId>
47.            <scope>provided</scope>
48.        </dependency>
49.        <dependency>
50.            <groupId>org.springframework.boot</groupId>
51.            <artifactId>spring-boot-starter-test</artifactId>
52.            <scope>test</scope>
53.        </dependency>

```

#### contenu du fichier pom.xml généré

```
54.     </dependencies>
55.
56.     <build>
57.         <plugins>
58.             <plugin>
59.                 <groupId>org.springframework.boot</groupId>
60.                 <artifactId>spring-boot-maven-plugin</artifactId>
61.             </plugin>
62.         </plugins>
63.     </build>
64.
65. </project>
```

#### Quelques explications :

La dépendance **spring-boot-starter-parent** permet de rapatrier la plupart des dépendances du projet. Sans elle, le fichier pom.xml serait plus complexe.

La dépendance **spring-boot-starter-web** indique à Spring Boot qu'il s'agit d'une application web, ce qui permet à Spring Boot de rapatrier les dépendances comme **SpringMVC**, **SpringContext**, et même le serveur d'application **Tomcat**, etc.

Vous avez aussi constaté la présence de la dépendance **spring-boot-starter-tomcat** qui n'est pas obligatoire, mais comme il s'agit d'une application web, Spring Boot par anticipation intègre un serveur d'application Tomcat afin de faciliter le déploiement de l'application. Cette dépendance n'étant pas obligatoire, vous pouvez la supprimer. Dans notre cas, on va utiliser un serveur Tomcat externe à travers une configuration Eclipse.

Il faut noter aussi l'absence des versions dans les dépendances. Ceci est dû au fait que Spring Boot gère de manière très autonome les versions et nous n'avons plus besoin de les déclarer.

Enfin, j'ai ajouté la déclaration ci-dessous qui permet de spécifier le **point d'entrée de l'application**. Cette déclaration est optionnelle, car Spring Boot est capable de rechercher tout seul la classe contenant la méthode **main(String[] args)**.

#### Spécification optionnelle du point d'entrée de l'application

```
<start-
class>com.bnngimgo.restclient.com.bnngimgo.springbootrestserver.SpringbootRestserverApplication</start-
class>
```



*Pour utiliser un serveur d'application différent, ou un serveur d'application externe, il faut commenter la dépendance Tomcat et déclarer juste la version cible dans la partie de déclaration des propriétés comme ci-dessous :*

```
1.
2.     <properties>
3.         <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
4.         <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
5.         <java.version>1.8</java.version>
6.         <tomcat.version>8.0.41</tomcat.version><!--remplace la version embarquée de Tomcat -->
7.     </properties>
```

La dépendance **spring-boot-starter-data-jpa** est nécessaire pour le développement de la couche de persistance. Les autres dépendances seront ajoutées notamment pour les besoins de tests.

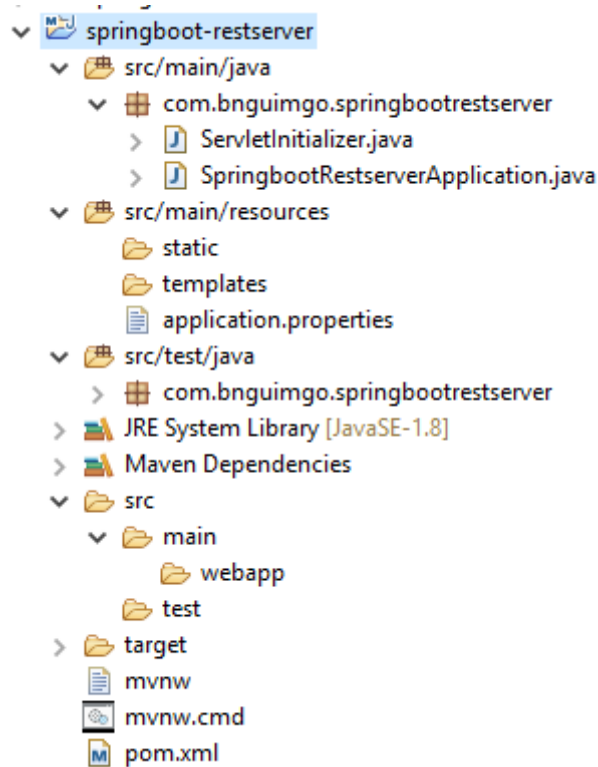


## I-B-1-b - Structure de l'application

Ci-dessous la structure du projet généré :



*Il n'est pas encore possible de tester l'application, car il manque un contrôleur pour envoyer ce qu'il faut afficher.*



Structure de l'application créée par génération



*Il faut relever particulièrement l'absence du fichier de descripteur du déploiement **web.xml** dont on a plus besoin dans une application Spring Boot. Voir ci-dessous les raisons.*

Le point d'entrée de l'application est la classe **SpringbootRestserverApplication**:

### Point d'entrée de l'application

```
1.
2. package com.bnguimgo.springbootrestserver;
3.
4. import org.springframework.boot.SpringApplication;
5. import org.springframework.boot.autoconfigure.SpringBootApplication;
6.
7. @SpringBootApplication
8. public class SpringbootRestserverApplication {
9.
10.     public static void main(String[] args) {
11.         SpringApplication.run(SpringbootRestserverApplication.class, args);
12.     }
13. }
```

L'application est initialisée par la classe **ServletInitializer**:

## Classe d'initialisation de l'application

```
1.
2. package com.bnguimbo.springbootrestserver;
3.
4. import org.springframework.boot.builder.SpringApplicationBuilder;
5. import org.springframework.boot.web.support.SpringBootServletInitializer;
6.
7. public class ServletInitializer extends SpringBootServletInitializer {
8.
9.     @Override
10.    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
11.        return application.sources(SpringbootRestserverApplication.class);
12.    }
13. }
```

Dans le package principal **com.bnguimbo.springbootrestserver**, nous avons les deux classes **ServletInitializer** et **SpringbootRestserverApplication**. Vous pouvez nommer ces classes comme vous voulez.

La classe **ServletInitializer** permet l'initialisation de l'application. Elle étend la classe **SpringBootServletInitializer** qui est à l'origine de cette initialisation et remplace ainsi l'ancien fichier **web.xml**.

La classe **SpringbootRestserverApplication** contient la méthode **void main(String[] args)** nécessaire dans une application Spring Boot, et permet l'exécution de celle-ci : c'est le point d'entrée de l'application.



*La classe de démarrage de l'application **SpringbootRestserverApplication** doit être à la racine du package principal si on veut permettre à Spring de scanner les sous-packages en utilisant l'annotation **@SpringBootApplication**.*



*L'annotation **@SpringBootApplication** est centrale dans une application Spring Boot et permet de scanner le package courant et ses sous-packages. Elle existe depuis **SpringBoot-1.2.0** et est équivalente à l'ensemble des annotations **@Configuration**, **@EnableAutoConfiguration** et **@ComponentScan***

### Voici les avantages qu'apporte cette annotation :

- Elle remplace l'annotation **@Configuration** qui permet de configurer une classe comme une source de définition des beans Spring.
- On aurait aussi pu ajouter l'annotation **@EnableWebMvc** pour indiquer qu'il s'agit d'une application SpringMVC, mais grâce à **@SpringBootApplication**, cette annotation n'est pas nécessaire, car Spring Boot va automatiquement l'ajouter dès qu'il verra que dans le workspace, il existe une librairie spring-webmvc.
- Elle remplace également l'annotation **@ComponentScan** qui autorise Spring à rechercher tous les composants, les configurations et autres services de l'application et à initialiser tous les contrôleurs.

Dans le dossier **src/main/resources**, il y a :

- Le dossier **static** qui permet de stocker tous les **fichiers images, CSS**, bref tous les fichiers qui ne fournissent pas un contenu dynamique (son utilisation n'est pas obligatoire).
- Le dossier **templates** qui permet de stocker des fichiers web si on utilise le **Framework Thymeleaf** de Spring (Thymeleaf ne sera pas utilisé dans le cadre de ce projet).
- Le fichier **application.properties** qui nous sera très utile pour configurer le projet. Pour le moment, il est vide, car je vais d'abord utiliser les configurations par défaut de Spring Boot.

## I-B-1-c - Création du service

Je vais créer un premier service REST qui permet juste de démarrer et tester l'application. Voici un petit service qui teste si le serveur a bien démarré (mettre cette classe dans le même package que **SpringbootRestserverApplication**) :

### Un mini service REST (RestServices) pour tester le démarrage de l'application

```

1.
2. package com.bnguimgo.springbootrestserver;
3. import org.slf4j.Logger;
4. import org.slf4j.LoggerFactory;
5. import org.springframework.http.HttpStatus;
6. import org.springframework.http.ResponseEntity;
7. import org.springframework.stereotype.Controller;
8. import org.springframework.web.bind.annotation.GetMapping;
9.
10. @Controller
11. public class RestServices {
12.
13.     private static final Logger logger = LoggerFactory.getLogger(RestServices.class);
14.
15.     @GetMapping(value = "/")
16.     public ResponseEntity<String> pong()
17.     {
18.         logger.info("Démarrage des services OK ....");
19.         return new ResponseEntity<String>("Réponse du serveur: "+HttpStatus.OK.name(),
20.             HttpStatus.OK);
21.     }
22. }
```



La classe est annotée **@Controller** afin de permettre à Spring d'enregistrer cette classe comme un contrôleur, et surtout de mémoriser les requêtes que cette classe est capable de gérer.

L'annotation **@GetMapping(value = "/")** est une nouvelle annotation introduite par Spring qui remplace l'annotation classique **@RequestMapping** et correspond exactement à **@RequestMapping(method=RequestMethod.GET, value = "/")**.

J'ai intégré un **logger** sans faire aucune configuration et ça marche. Si vous regardez votre console, vous verrez bien les traces de logs lors du démarrage de l'application. C'est l'un des avantages de Spring Boot qui configure par défaut un logger pour nous. Je vais vous montrer plus loin comment personnaliser le logger par défaut. Le service est prêt à être testé.

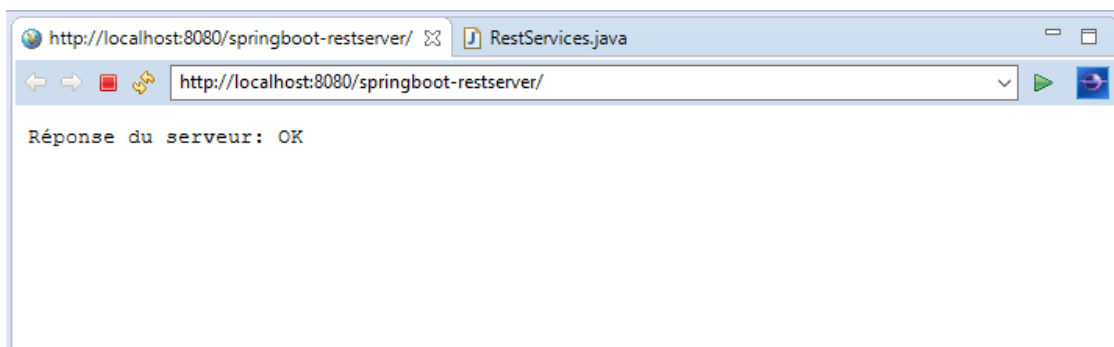
## I-B-1-d - Test de la configuration

Le service sera testé avec un serveur d'application externe Tomcat. Il faut donc configurer ce serveur dans l'IDE Eclipse. Pour ajouter un nouveau serveur dans Eclipse :

Fichier --> Nouveau Others... --> Server --> Next --> Tomcat v8.0 Server --> Finish

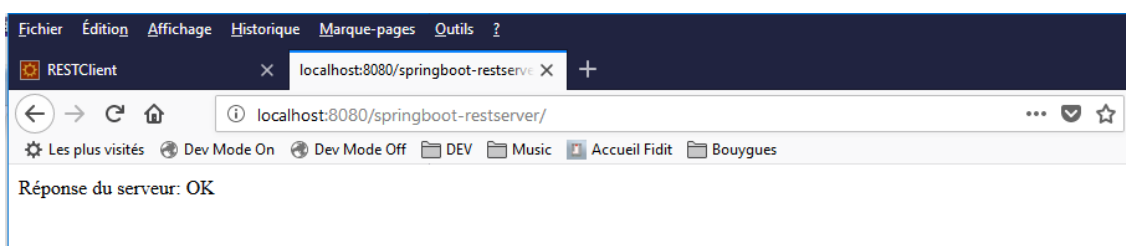
Pour tester l'application dans Eclipse, après compilation : clic droit sur l'application --> Run As --> Run on Server --> Tomcat v8.0 Server at localhost --> Next --> Finish

Résultat dans Eclipse:



### Test du service REST dans l'IDE Eclipse

Il est possible de tester directement sur un navigateur web ou un client REST. Voici le résultat dans le navigateur Mozilla:



### Test de l'application sur Mozilla

Maintenant que notre environnement est en place, on peut développer un service REST complet.



*Je rappelle que dans cette partie, seul le serveur est développé, mais le client sera développé aussi dans la deuxième partie du tutoriel. Le travail à faire consiste à développer un service qui permet à un internaute de créer son compte et de se connecter à l'application, le tout en services REST.*

## I-B-2 - Création des couches

Dans cette partie, je vais développer un service complet de gestion d'un utilisateur :

- Extraction de tous les utilisateurs.
- Extraction d'un utilisateur à base de son identifiant.
- Création d'un utilisateur.
- Mise à jour de l'utilisateur.
- Suppression d'un utilisateur.

### I-B-2-a - Création du model

Pour mettre en place le service d'extraction, il nous faut un objet utilisateur (**User**). Nous aimerions aussi connaître les rôles de chaque utilisateur, par exemple s'il est administrateur (**ROLE\_ADMIN**) ou simple utilisateur (**ROLE\_USER**) de l'application. Il faudra donc aussi créer un objet Role pour stocker ces informations. Sachant qu'un utilisateur peut avoir les deux rôles, cette relation donnera lieu à une nouvelle association **ManyToMany**. Voyons toute de suite comment implémenter.

Il faut créer un nouveau sous-package nommé **model** dans le package racine **com.nguimgo.springbootrestserver**. Il faut ajouter dans le model les deux entités ci-dessous :

## Création de l'entité User

```

1.
2. package com.bnguimbo.springbootrestserver.model;
3.
4. import java.io.Serializable;
5. import java.util.HashSet;
6. import java.util.Set;
7.
8. import javax.persistence.*;
9. import javax.xml.bind.annotation.XmlElement;
10. import javax.xml.bind.annotation.XmlRootElement;
11.
12. import com.bnguimbo.springbootrestserver.dto.UserDTO;
13. import com.bnguimbo.restclient.dto.UserRegistrationForm;
14.
15. @Entity
16. @Table(name = "UTILISATEUR")
17. @XmlRootElement(name = "user")
18. public class User implements Serializable{
19.
20.     @Id
21.     @GeneratedValue(strategy = GenerationType.IDENTITY)
22.     @Column(name = "USER_ID", updatable = false, nullable = false)
23.     private Long id;
24.
25.     @Column(name = "LOGIN", unique=true, insertable=true, updatable=true, nullable=false)
26.     private String login;
27.
28.     @Column(name = "USER_PASSWORD", insertable=true, updatable=true, nullable=false)
29.     private String password;
30.
31.     @Column(name = "USER_ACTIVE", insertable=true, updatable = true, nullable=false)
32.     private Integer active;
33.
34.     @ManyToMany(cascade = CascadeType.DETACH)
35.     @JoinTable(name = "USER_ROLE", joinColumns = @JoinColumn(name = "USER_ID"),
36.         inverseJoinColumns = @JoinColumn(name = "ROLE_ID"))
37.     private Set<Role> roles= new HashSet<>();
38.
39.     public User() {
40.         super();
41.     }
42.
43.     public User(String login, String password, Integer active) {
44.         this.login = login;
45.         this.password = password;
46.         this.active = active;
47.     }
48.
49.     public User(Long id, String login) {
50.         this.id = id;
51.         this.login = login;
52.     }
53.
54.     public User(String login) {
55.         this.login = login;
56.     }
57.
58.     public User(UserDTO userDTO) {
59.         this.setId(userDTO.getId());
60.         this.setLogin(userDTO.getLogin());
61.         this.setPassword(userDTO.getPassword());
62.     }
63.
64.     public User(UserRegistrationForm userRegistrationForm) {
65.         this.setLogin(userRegistrationForm.getLogin());
66.         this.setPassword(userRegistrationForm.getPassword());
67.     }
68.
69.     public User(Long id, String login, String password, Integer active) {
70.         this.id= id;
71.         this.login=login;

```

## Création de l'entité User

```

71.         this.password = password;
72.         this.active=active;
73.     }
74.
75.     public Long getId() {
76.         return id;
77.     }
78.
79.     @XmlElement
80.     public void setId(Long id) {
81.         this.id = id;
82.     }
83.
84.     public String getLogin() {
85.         return login;
86.     }
87.     @XmlElement
88.     public void setLogin(String login) {
89.         this.login = login;
90.     }
91.
92.     public String getPassword() {
93.         return password;
94.     }
95.
96.     @XmlElement
97.     public void setPassword(String password) {
98.         this.password = password;
99.     }
100.
101.     public Integer getActive() {
102.         return active;
103.     }
104.     @XmlElement
105.     public void setActive(Integer active) {
106.         this.active = active;
107.     }
108.
109.     public Set<Role> getRoles() {
110.         return roles;
111.     }
112.
113.     @XmlElement
114.     public void setRoles(Set<Role> roles) {
115.         this.roles = roles;
116.     }
117.
118.     @Override
119.     public String toString() {
120.         return "User [id=" + id + ", login=" + login + ", password=XXXXXXX, active=" +
active + ", roles="
121.             + roles + "]\n";
122.     }
123.
124.     @Override
125.     public int hashCode() {
126.         final int prime = 31;
127.         int result = 1;
128.         result = prime * result + ((active == null) ? 0 : active.hashCode());
129.         result = prime * result + ((id == null) ? 0 : id.hashCode());
130.         result = prime * result + ((login == null) ? 0 : login.hashCode());
131.         result = prime * result + ((password == null) ? 0 : password.hashCode());
132.         result = prime * result + ((roles == null) ? 0 : roles.hashCode());
133.         return result;
134.     }
135.
136.     @Override
137.     public boolean equals(Object obj) {
138.         if (this == obj)
139.             return true;

```

## Création de l'entité User

```

140.         if (obj == null)
141.             return false;
142.         if (getClass() != obj.getClass())
143.             return false;
144.         User other = (User) obj;
145.         if (active == null) {
146.             if (other.active != null)
147.                 return false;
148.         } else if (!active.equals(other.active))
149.             return false;
150.         if (id == null) {
151.             if (other.id != null)
152.                 return false;
153.         } else if (!id.equals(other.id))
154.             return false;
155.         if (login == null) {
156.             if (other.login != null)
157.                 return false;
158.         } else if (!login.equals(other.login))
159.             return false;
160.         if (password == null) {
161.             if (other.password != null)
162.                 return false;
163.         } else if (!password.equals(other.password))
164.             return false;
165.         if (roles == null) {
166.             if (other.roles != null)
167.                 return false;
168.         } else if (!roles.equals(other.roles))
169.             return false;
170.         return true;
171.     }
172. }

```

Création d'un **UserDTO** pour stocker les données utilisateurs qui transitent sur le réseau ou entre les couches de l'application. Pour une bonne organisation du code, il faut créer cet objet dans un nouveau package nommé dto.

## UserDTO

```

1.
2. package com.bnguimgo.springbootrestserver.dto;
3. import java.io.Serializable;
4.
5. public class UserDTO implements Serializable{
6.
7.     private static final long serialVersionUID = -443589941665403890L;
8.
9.     private Long id;
10.
11.     private String login;
12.     private String password;
13.     private String userType;
14.
15.     public UserDTO() {
16.     }
17.
18.     public UserDTO(String login, String password) {
19.         this.login = login;
20.         this.password = password;
21.     }
22.
23.     public UserDTO(String login, String password, String userType) {
24.         this.login = login;
25.         this.password = password;
26.         this.userType = userType;
27.     }
28.
29.     public UserDTO(Long id, String login) {
30.         this.id = id;

```

## UserDTO

```

31.         this.login = login;
32.     }
33.
34.     public UserDTO(Long id, String login, String password, String userType) {
35.         this.id = id;
36.         this.login = login;
37.         this.password = password;
38.         this.userType = userType;
39.     }
40.
41.     public Long getId() {
42.         return id;
43.     }
44.
45.     public void setId(Long id) {
46.         this.id = id;
47.     }
48.
49.     public String getLogin() {
50.         return login;
51.     }
52.
53.     public void setLogin(String login) {
54.         this.login = login;
55.     }
56.
57.     public String getPassword() {
58.         return password;
59.     }
60.
61.     public void setPassword(String password) {
62.         this.password = password;
63.     }
64.
65.     public String getUserType() {
66.         return userType;
67.     }
68.
69.     public void setUserType(String userType) {
70.         this.userType = userType;
71.     }
72.
73.     @Override
74.     public String toString() {
75.         return String.format("[id=%s, mail=%s, userType=%s]", id, login, userType);
76.     }
77.
78. }

```

L'annotation **@Entity** permet d'indiquer que cette classe sera une table de la base de données et l'annotation **@Table(name = "UTILISATEUR")** permet de donner le nom UTILISATEUR à la table. Grâce à ces annotations, on n'a plus besoin du fichier de configuration **persistence.xml**.



*Il faut éviter de nommer ou de créer une table avec pour nom **USER**, car dans la plupart des bases de données, il y a déjà une table utilisateur nommée **USER**.*

L'annotation **@XmlRootElement(name = "user")** permettra de construire un objet XML lors des tests de communications entre le client et le serveur. Voici **un exemple d'objet XML** qu'on pourra utiliser lors des tests pour créer un utilisateur :

## exemple d'objet XML à envoyer au serveur

```

1.
2. <user>
3.   <email>admin4@admin4.com</email>

```



#### exemple d'objet XML à envoyer au serveur

```

4.    <password>admin4</password>
5.    <active>1</active>
6.    <roles>
7.      <role>
8.        <roleName>ROLE_ADMIN</roleName>
9.      </role>
10.     <role>
11.       <roleName>ROLE_USER</roleName>
12.     </role>
13.   </roles>
14. </user>
  
```

Une personne peut à la fois avoir le rôle administrateur et le rôle utilisateur. Plusieurs utilisateurs peuvent avoir le même rôle. On peut par exemple avoir plusieurs administrateurs : c'est ce qui explique l'existence de la relation **ManyToMany**. Au niveau de la base de données, cela donne naissance à une table de jointure que je vais nommer : **USER\_ROLE**, à ne pas confondre avec la table **ROLE** qui stocke tous les rôles. On a ainsi terminé le model.

### I-B-2-b - Création de la base de données

Dans cette section, on va **initialiser la base de données par des scripts** et se connecter dessus sans faire aucune déclaration ni aucune configuration de la **datasource**. *C'est un des avantages de Spring Boot.*

Il faut remarquer que jusqu'à présent, il n'y a eu aucune configuration concernant **Hibernate** pour le mapping objet relationnel. Le simple fait d'avoir déclaré une base de données **H2** dans le **pom.xml** fait que Spring Boot configure **HibernateEntityManager** pour nous. Grâce aux annotations appliquées sur chaque objet model, Hibernate va transformer chaque entité ou association en une table et créer dans la foulée la base de données.

Voici le script de l'initialisation de la base de données nommée **data.sql**. Ce script doit être déposé dans le répertoire de gestion des ressources **src/main/resources**. Spring Boot va automatiquement l'utiliser pour initialiser la base de données au démarrage de l'application, pas besoin de déclarer une datasource.

#### Script d'initialisation de la base de données data.sql

```

1.
2. --INITIALISATION TABLE ROLE
3. INSERT INTO ROLE(ROLE_ID,ROLE_NAME) VALUES (1,'ROLE_ADMIN');
4. INSERT INTO ROLE(ROLE_ID,ROLE_NAME) VALUES (2,'ROLE_USER');
5.
6. --INITIALISATION TABLE UTILISATEURS
7. INSERT INTO UTILISATEUR(USER_ID, LOGIN, USER_PASSWORD,
8.   USER_ACTIVE) values (1, 'admin', 'admin', 1);
9. INSERT INTO UTILISATEUR(USER_ID, LOGIN, USER_PASSWORD,
10.  USER_ACTIVE) values (2, 'user', 'user', 1);
11. INSERT INTO UTILISATEUR(USER_ID, LOGIN, USER_PASSWORD,
12.  USER_ACTIVE) values (3, 'user1', 'user1', 0);--inactif user
13.
14. -- TABLE DE JOINTURE
15. INSERT INTO USER_ROLE(USER_ID,ROLE_ID) VALUES (1,1);
16. INSERT INTO USER_ROLE(USER_ID,ROLE_ID) VALUES (1,2);
17. INSERT INTO USER_ROLE(USER_ID,ROLE_ID) VALUES (2,2);
18. INSERT INTO USER_ROLE(USER_ID,ROLE_ID) VALUES (3,2);
19.
20. COMMIT;
  
```

Notez bien la table d'association (**USER\_ROLE**) qui fait le lien **ManyToMany** décrit plus haut.

## I-B-2-c - Création de la couche DAO



*Le développement de la couche DAO et même de la couche de services ne concerne pas spécifiquement les services REST. La partie spécifique aux services REST est mise en place au niveau des contrôleurs que nous allons voir plus loin.*

La dépendance **spring-boot-starter-data-jpa** que nous avons ajoutée dans le pom.xml permet d'utiliser **SpringData** qui est une implémentation de JPA (Java Persistence Api). SpringData implémente toutes méthodes du CRUD (**Create, Read, Update, Delete**) quand on hérite de JpaRepository ou de CrudRepository.

Interface UserRepository (à créer dans un nouveau package nommé dao) :

```
1.
2. package com.bnguimgo.springboot.rest.server.dao;
3.
4. import org.springframework.data.jpa.repository.JpaRepository;
5.
6. import com.bnguimgo.springboot.rest.server.entities.User;
7.
8. public interface UserRepository extends JpaRepository<User, Long> {
9.
10.     User findByLogin(String login);
11.
12. }
```

On peut ajouter dans l'interface des méthodes qui ne font pas partie du CRUD. C'est le cas de la méthode **findByLogin(String login)**



*Remarque : pas besoin d'implémenter les méthodes **createUser(User user)**, **deleteUser(Long id)**, etc. À l'exécution, SpringData va créer automatiquement une implémentation de cette interface, et par conséquent, pas besoin d'une classe d'implémentation nommée **UserRepositoryImpl**.*

Voici l'interface Role

```
1.
2. package com.bnguimgo.springbootrestserver.dao;
3. import java.util.stream.Stream;
4.
5. import org.springframework.data.jpa.repository.JpaRepository;
6. import org.springframework.data.jpa.repository.Query;
7.
8. import com.bnguimgo.springbootrestserver.model.Role;
9.
10. public interface RoleRepository extends JpaRepository<Role, Long> {
11.
12.     Role findByRoleName(String roleName);
13.
14.     @Query("select role from Role role")
15.     Stream<Role> getAllRolesStream(); // Java8 Stream : on place la liste des rôles dans un Stream
16. }
```

## I-B-2-d - Couche de services

Interface UserService créée dans un nouveau package nommé service

```
1.
2. package com.bnguimgo.springbootrestserver.service;
3. import java.util.Collection;
4.
```

### Interface UserService créée dans un nouveau package nommé service

```
5. import com.bnguimgo.springbootrestserver.model.User;
6.
7. public interface UserService {
8.
9.     Collection<User> getAllUsers();
10.
11.     User getUserById(Long id);
12.
13.     User findByLogin(String login);
14.
15.     User saveOrUpdateUser(User user);
16.
17.     void deleteUser(Long id);
18.
19. }
```

### Ajoutez l'interface RoleService dans le même package

```
1.
2. package com.bnguimgo.springbootrestserver.service;
3. import java.util.Collection;
4. import java.util.stream.Stream;
5.
6. import com.bnguimgo.springbootrestserver.model.Role;
7.
8. public interface RoleService {
9.
10.     Role findByName(String roleName);
11.
12.     Collection<Role> getAllRoles();
13.
14.     Stream<Role> getAllRolesStream();
15. }
```

Les deux méthodes **getAllRoles()** et **getAllRolesStream()** font la même chose. Je veux juste montrer à travers ces deux méthodes comment Java8 permet d'implémenter différemment les collections.

### Implémentation des services :

Le mot de passe utilisateur sera **encrypté** afin d'éviter de le stocker en clair dans la base de données. Pour le faire, je vais utiliser la dépendance **spring-boot-starter-security** que je vous invite à ajouter dans pom.xml.

### ajout de la dépendance spring-boot-starter-security

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Pour utiliser les Collections Apache, il faut ajouter la dépendance ci-dessous :

### ajout de la dépendance apache commons-collections4

```
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-collections4</artifactId>
  <version>4.0</version>
</dependency>
```

Je vais créer aussi une classe de configuration des beans nommée **BeanConfiguration** qui va permettre de déclarer des beans. Par exemple le bean **BCryptPasswordEncoder**, ce qui permettra à spring de créer une instance de cet objet pour nous. Je vais ensuite utiliser cet objet pour hacher les mots de passe lors de la persistance d'un objet.

### Création de la classe BeanConfiguration dans un nouveau package component

```

1.
2. package com.bnngimgo.springbootrestserver.component;
3. import org.springframework.context.annotation.Bean;
4. import org.springframework.context.annotation.Configuration;
5. import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
6. import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;
7.
8. @Configuration
9. public class BeanConfiguration extends WebMvcConfigurerAdapter {
10.
11.     @Bean
12.     public BCryptPasswordEncoder passwordEncoder() {
13.         BCryptPasswordEncoder bCryptPasswordEncoder = new BCryptPasswordEncoder();
14.         return bCryptPasswordEncoder;
15.     }
16.
17. }
  
```

L'annotation **@Configuration** indique à Spring que cette classe est une source de configuration des beans. Dans cette classe, j'ai déclaré un seul bean avec l'annotation **@Bean**. On pourra ainsi créer une instance de **BCryptPasswordEncoder** en appliquant l'annotation **@Autowired** de Spring partout où on en a besoin.

### Implémentation de l'interface UserService avec encryptage du mot de passe utilisateur

```

1.
2. package com.bnngimgo.springbootrestserver.service;
3. import java.util.Collection;
4.
5. import org.apache.commons.collections4.IteratorUtils;
6. import org.springframework.beans.factory.annotation.Autowired;
7. import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
8. import org.springframework.stereotype.Service;
9.
10. import com.bnngimgo.springbootrestserver.dao.UserRepository;
11. import com.bnngimgo.springbootrestserver.model.User;
12.
13. @Service(value = "userService") // l'annotation @Service est optionnelle ici, car il n'existe qu'une seule imp
14. public class UserServiceImpl implements UserService {
15.
16.     @Autowired
17.     private UserRepository userRepository;
18.
19.     @Autowired
20.     private BCryptPasswordEncoder bCryptPasswordEncoder;
21.
22.     @Override
23.     public User findByLogin(String login) {
24.         return userRepository.findByLogin(login);
25.     }
26.
27.     @Override
28.     public Collection<User> getAllUsers() {
29.         return IteratorUtils.toList(userRepository.findAll().iterator());
30.     }
31.
32.     @Override
33.     public User getUserById(Long id) {
34.         return userRepository.findOne(id);
35.     }
36.
37.     @Override
38.     @Transactional(readOnly=false)
39.     public User saveOrUpdateUser(User user) {
40.         user.setPassword(bCryptPasswordEncoder.encode(user.getPassword()));
41.         return userRepository.save(user);
42.     }
43.
44.     @Override
45.     @Transactional(readOnly=false)
46.     public void deleteUser(Long id) {
  
```

### Implémentation de l'interface UserService avec encryptage du mot de passe utilisateur

```
47.         userRepository.delete(id);
48.
49.     }
50.
51. }
```

L'annotation **@Service(value = "userService")** permet de déclarer cette classe comme un bean de service. Il faut noter également l'injection des dépendances par l'annotation **@Autowired** de deux instances d'objets (**userRepository**, **BCryptPasswordEncoder**) créées par Spring. L'objet **userRepository** permet d'avoir accès aux services de la couche **DAO** (services fournis par l'implémentation Spring de **JpaRepository**). L'objet **BCryptPasswordEncoder** permet de hacher les mots de passe pour ne pas les stocker de manière visible au niveau de la base de données.

### Implémentation de l'interface RoleService pour gérer les rôles utilisateur

```
1.
2. package com.bnguimgo.springbootrestserver.service;
3. import java.util.Collection;
4. import java.util.stream.Stream;
5.
6. import org.apache.commons.collections4.IteratorUtils;
7. import org.springframework.beans.factory.annotation.Autowired;
8. import org.springframework.stereotype.Service;
9.
10. import com.bnguimgo.springbootrestserver.dao.RoleRepository;
11. import com.bnguimgo.springbootrestserver.model.Role;
12.
13.
14. @Service(value = "roleService")
15. public class RoleServiceImpl implements RoleService {
16.
17.     @Autowired
18.     private RoleRepository roleRepository;
19.
20.     @Override
21.     public Collection<Role> getAllRoles() { //Avant JAVA8
22.         return IteratorUtils.toList(roleRepository.findAll().iterator());
23.     }
24.
25.     @Override
26.     public Stream<Role> getAllRolesStream() { //JAVA8
27.         return roleRepository.getAllRolesStream();
28.     }
29.
30.     @Override
31.     public Role findByName(String roleName) {
32.         return roleRepository.findByName(roleName);
33.     }
34. }
```

Notez bien l'implémentation différente des deux méthodes qui renvoient les **collections** avec **getAllRoles()** et le **Stream** avec **getAllRolesStream()**.

## I-B-3 - Mise en place du service REST

Il s'agit de développer dans cette partie la couche de contrôle. C'est cette couche qui intercepte et filtre toutes les requêtes utilisateurs. Chaque contrôleur dispose d'un service pour traiter les requêtes: c'est le service REST.

### I-B-3-a - Création du filtre Cross Domain

Je vous propose de créer la **classe utilitaire CrossDomainFilter** qui a pour objectif de pallier les problèmes de Cross-Domain. En fait, comme le client et le serveur peuvent être hébergés sur deux serveurs distants, il faut penser

aux problèmes réseau qui peuvent entraver la communication. Il faut indiquer au serveur quels sont les types d'entêtes HTTP à prendre en considération. Plus d'explications sur les filtres [Cross-Domain](#) ici:

Classe utilitaire nommée `CrossDomainFilter` à ajouter dans le package `component`

```
1.
2. package com.bnguimgo.springbootrestserver.component;
3. import org.springframework.stereotype.Component;
4. import org.springframework.web.filter.OncePerRequestFilter;
5.
6. import javax.servlet.FilterChain;
7. import javax.servlet.ServletException;
8. import javax.servlet.http.HttpServletRequest;
9. import javax.servlet.http.HttpServletResponse;
10. import java.io.IOException;
11.
12. @Component
13. public class CrossDomainFilter extends OncePerRequestFilter {
14.     @Override
15.     protected void doFilterInternal(HttpServletRequest httpServletRequest,
16.         HttpServletResponse httpServletResponse, FilterChain filterChain)
17.         throws ServletException, IOException {
18.         httpServletResponse.addHeader("Access-Control-Allow-Origin", "*"); //toutes les URI sont autorisées
19.         httpServletResponse.addHeader("Access-Control-Allow-Methods", "GET, POST, PUT, DELETE, OPTIONS");
20.         httpServletResponse.addHeader("Access-Control-Allow-Headers", "origin, content-type, accept, x-req");
21.         filterChain.doFilter(httpServletRequest, httpServletResponse);
22.     }
23. }
```

Grâce à l'annotation **@Component**, Spring va considérer cette classe comme un composant et pourra utiliser le filtre sans problème.

Une dernière configuration concerne les fichiers `application.properties`. J'ai mis à disposition trois fichiers de configuration :

- un fichier **application-dev.properties** à utiliser en phase de développement ;
- un fichier **application-prod.properties** à utiliser en phase de production ;
- un fichier **application.properties** pour initier toutes les configurations de base.

En ajoutant par exemple **spring.profiles.active=dev** dans `application.properties`, Spring Boot sait que c'est le profil **dev** qui est activé et ce sont les fichiers de configuration **application-dev.properties** et **application.properties** qui seront chargés. Je vous invite à regarder la configuration de chaque fichier.

Pour désactiver la demande récurrente du mot de passe généré par Spring security en phase de développement, il faut ajouter cette déclaration **security.basic.enabled=false** dans le fichier `application.properties`.

Voici le contenu de chacun de ces trois fichiers:

Le fichier **application.properties** est le fichier de configuration principale

`application.properties`

```
1.
```

### application.properties

```

2. #spring.profiles.active=dev
3. spring.profiles.active=prod
4. security.basic.enabled=false
5.
6. spring.h2.console.enabled=true
7. spring.h2.console.path=/console
  
```

On constate bien que c'est le profil de production (**prod**) qui est activé, ce qui veut dire que c'est le fichier de propriétés **application-prod.properties** qui sera chargé.

### application-prod.properties

```

1.
2. #Charge les proprietes de la production
3. nextpage.message=Salut Vous etes en production
4. error.no.user.id = Aucun utilisateur avec l'identifiant:
5. error.no.resource = Not found
6. technical.error = Erreur technique !!!
7.
8. -Dmaven.test.skip=true
9.
10. ##### BASE DE DONNEES #####
11. logging.level.org.hibernate.SQL=error
12. # Supprime et re-crée les tables et sequences existantes , charge le script
    d'initialisation de la base de données data.sql
13. spring.jpa.hibernate.ddl-auto=create-drop
14.
15. ##### GESTION DES LOGS #####
16. logging.level.org.springframework.web=DEBUG
17. logging.level.com.bnguimgo.springboot.rest.server=DEBUG
18. #
19. # Pattern impression des logs console
20. logging.pattern.console= %d{yyyy-MM-dd HH:mm:ss} - %msg%n
21.
22. # Pattern impression des logs dans un fichier
23. logging.pattern.file= %d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg%n
24. # Redirection des logs vers un fichier du repertoire Temp, exemple sur windows: C:\Users
    \UserName\AppData\Local\Temp\
25. logging.file=${java.io.tmpdir}/logs/restServer/applicationRestServer.log
  
```

Vous pouvez modifier le paramétrage de ce fichier à votre souhait pour, par exemple, mieux gérer les logs. On remarque qu'avec le profil de production, les tests sont désactivés par le paramétrage : **-Dmaven.test.skip=true**

### application-dev.properties

```

1.
2. #Chargement des propriétés ci-dessous au démarrage de l'application
3. nextpage.message=Salut vous etes en profile dev sur Rest Server
4.
5. ##### Configuration des Logs #####
6. logging.level.root= WARN
7. logging.level.org.springframework.security= DEBUG
8. logging.level.org.springframework.web= ERROR
9. logging.level.org.apache.commons.dbcp2= DEBUG
10.
11. # Pattern impression des logs console
12. logging.pattern.console= %d{yyyy-MM-dd HH:mm:ss} - %msg%n
13.
14. # Pattern impression des logs dans un fichier
15. logging.pattern.file= %d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg%n
16. # Redirection des logs vers un fichier du repertoire Temp, exemple sur windows: C:\Users
    \UserName\AppData\Local\Temp\
17. logging.file=${java.io.tmpdir}/logs/rest/applicationRestServer.log
18.
19. ##### BASE DE DONNEES #####
20. logging.level.org.hibernate.SQL=debug
21. # Supprime et recrée les tables et séquences existantes , exécute le script data.sql qui
    initialise la base de données
22. spring.jpa.hibernate.ddl-auto=create-drop
  
```



Il faut noter que jusqu'à présent, nous n'avons développé aucune page JSP, et par conséquent il n'y a rien à mettre dans le dossier webapp, ce qui est normal, car dans cette partie, nous voulons juste exposer des services. **La création d'un service REST se passe globalement dans les contrôleurs** comme nous allons le voir à travers les implémentations ci-dessous.

### I-B-3-b - Nouveautés dans les requêtes HTTP avec Spring

**Spring-4.3** a introduit de nouvelles annotations qui facilitent les requêtes HTTP dans des projets SpringMVC. Spring prend actuellement en charge cinq nouveaux types d'annotations intégrées pour la gestion de différents types de requêtes HTTP : **GET, POST, PUT, DELETE** et **PATCH** (mise à jour partielle).

Ces annotations sont:

- 1 @GetMapping = @RequestMapping + Http GET method ;
- 2 @PostMapping = @RequestMapping + Http POST method ;
- 3 @PutMapping = @RequestMapping + Http PUT method ;
- 4 @DeleteMapping = @RequestMapping + Http DELETE method.

#### Exemple de méthode HTTP GET avec la nouvelle annotation

```
1.
2. @GetMapping("/user/{userId}")
3. public User findUserById (@PathVariable Long userId) {
4.     return userService.findUserById(userId);
5. }
```


**Avant la nouvelle annotation** : en règle générale, si nous annotons une méthode en utilisant l'annotation traditionnelle @RequestMapping, on ferait quelque chose comme ceci :

#### Exemple de méthode HTTP GET avec l'ancienne annotation

```
1.
2. @RequestMapping(method=RequestMethod.GET, value = "/user/{userId}", produces =
   MediaType.APPLICATION_JSON_VALUE)
3. public User findUserById(@PathVariable Long userId) {
4.     return userService.findUserById(userId);
5. }
```

Nous avons spécifié le type de média JSON que le client devrait être capable de traiter. Les avantages des nouvelles annotations résident dans le fait qu'on peut indifféremment produire du XML ou du JSON sans se préoccuper du type de média. Pour avoir cette flexibilité, il est donc conseillé de ne pas spécifier le type de média, et d'utiliser les nouvelles annotations qu'on vient de décrire.

### I-B-3-c - Création d'un contrôleur par défaut

Nous avons déjà créé dans la section  **I-B-1-c** un petit service qui permettait de tester la configuration de notre environnement. Je vous propose de renommer cette classe (**RestServices**) en **DefaultController** et de la déplacer dans un nouveau package appelé **controller**. Je vous remets le résultat ici :

#### contrôleur par défaut DefaultController

```
1.
2. package com.bnguimgo.springbootrestserver.controller;
3. import org.slf4j.Logger;
4. import org.slf4j.LoggerFactory;
5. import org.springframework.http.HttpStatus;
6. import org.springframework.http.ResponseEntity;
7. import org.springframework.stereotype.Controller;
8. import org.springframework.web.bind.annotation.GetMapping;
9. @Controller
10. public class DefaultController {
```



### contrôleur par défaut DefaultController

```

11.     private static final Logger logger = LoggerFactory.getLogger(DefaultController.class);

12.     @GetMapping(value = "/")
13.     public ResponseEntity<String> pong() {
14.         logger.info("Démarrage des services OK .....");
15.         return new ResponseEntity<String>("Réponse du serveur: "+HttpStatus.OK.name(),
            HttpStatus.OK);
16.     }
17. }

```

La classe **DefaultController** est juste un contrôleur utilitaire qui va nous renseigner automatiquement dès le démarrage du client **si le serveur REST est à l'écoute** grâce à la réponse renvoyée par **HttpStatus**. Si le serveur est actif, le message de réponse sera: **Réponse du serveur: OK**, avec le code de réponse **HTTP = 200**. Sinon, une page d'erreur sera affichée.

L'utilisation de la classe **ResponseEntity** de Spring permet de prendre en compte la gestion des **statuts HTTP** de réponses. On peut toutefois utiliser la classe **ResponseBody** pour traiter les réponses si on n'a pas besoin d'exploiter les codes de réponses HTTP. Dans notre cas, je préfère **ResponseEntity**. On peut donc dire que **ResponseEntity = ResponseBody + HttpStatus**.

Le service est désormais prêt et vous pouvez le tester. Pour tester l'application après l'avoir déployée sous **Tomcat**, voici l'URL de test : <http://localhost:8080/springboot-restserver/>.

On obtient le même résultat que lors du premier test de la configuration vu précédemment dans la section **I-B-1-c**. N'hésitez pas à regarder de temps en temps les **logs** dans la console. Dans la classe *UserController* ci-dessous, je vais développer les services de **lecture, de création, de mise à jour et de suppression** d'un utilisateur.

### I-B-3-d - Service d'extraction de tous les utilisateurs

#### service REST d'extraction de tous les utilisateurs dans UserController

```

1.
2. package com.bnngimgo.springbootrestserver.controller;
3.
4. import java.util.Collection;
5. import java.util.HashSet;
6. import java.util.Set;
7. import java.util.stream.Collectors;
8. import java.util.stream.Stream;
9.
10. import org.slf4j.Logger;
11. import org.slf4j.LoggerFactory;
12. import org.springframework.beans.factory.annotation.Autowired;
13. import org.springframework.http.HttpStatus;
14. import org.springframework.http.ResponseEntity;
15. import org.springframework.stereotype.Controller;
16. import org.springframework.transaction.annotation.Transactional;
17. import org.springframework.web.bind.annotation.DeleteMapping;
18. import org.springframework.web.bind.annotation.GetMapping;
19. import org.springframework.web.bind.annotation.PathVariable;
20. import org.springframework.web.bind.annotation.PostMapping;
21. import org.springframework.web.bind.annotation.PutMapping;
22. import org.springframework.web.bind.annotation.RequestBody;
23. import org.springframework.web.bind.annotation.RequestMapping;
24.
25. import com.bnngimgo.springbootrestserver.model.Role;
26. import com.bnngimgo.springbootrestserver.model.User;
27. import com.bnngimgo.springbootrestserver.service.RoleService;
28. import com.bnngimgo.springbootrestserver.service.UserService;
29.
30. @Controller
31. @CrossOrigin(origins = "http://localhost:8080", maxAge = 3600)
32. @RequestMapping("/user/*")
33. public class UserController {

```

### service REST d'extraction de tous les utilisateurs dans UserController

```

34.
35.     private static final Logger logger = LoggerFactory.getLogger(UserController.class);
36.
37.     @Autowired
38.     private UserService userService;
39.     @Autowired
40.     private RoleService roleService;
41.
42.     @GetMapping(value = "/users")
43.     public ResponseEntity<Collection<User>> getAllUsers() {
44.         Collection<User> users = userService.getAllUsers();
45.         logger.info("liste des utilisateurs : " + users.toString());
46.         return new ResponseEntity<Collection<User>>(users, HttpStatus.FOUND);
47.     }
48. }
  
```

L'annotation **@CrossOrigin(origins = "http://localhost:8080", maxAge = 3600)** permet de favoriser une communication distante entre le client et le serveur, c'est-à-dire lorsque le client et le serveur sont déployés dans deux serveurs distincts, ce qui permet d'éviter des problèmes réseau.

Compiler et déployer l'application dans Tomcat-8. Vous pouvez utiliser n'importe quel client REST pour faire les tests. J'utilise **RESTClient** ou **Boomerang**. Voici l'URI qui ramène tous les utilisateurs : **<http://localhost:8080/springboot-restserver/user/users>**

### Résultat de la requête HTTP GET

```

1.
2. [{
3.   "id": 1,
4.   "login": "admin",
5.   "password": "admin",
6.   "active": 1,
7.   "roles": [{
8.     "id": 2,
9.     "roleName": "ROLE_USER"
10.  }, {
11.    "id": 1,
12.    "roleName": "ROLE_ADMIN"
13.  }]
14. }, {
15.   "id": 2,
16.   "login": "user2",
17.   "password": "user2",
18.   "active": 1,
19.   "roles": [{
20.     "id": 2,
21.     "roleName": "ROLE_USER"
22.   }]
23. }, {
24.   "id": 3,
25.   "login": "user3",
26.   "password": "user3",
27.   "active": 0,
28.   "roles": [{
29.     "id": 2,
30.     "roleName": "ROLE_USER"
31.   }]
32. }]
  
```

## I-B-3-e - Service création d'un utilisateur

### Service de création d'un utilisateur à ajouter dans la classe UserController

```

1.
2.     @PostMapping(value = "/users")
3.     @Transactional
4.     public ResponseEntity<User> saveUser(@RequestBody User user) {
  
```

### Service de création d'un utilisateur à ajouter dans la classe UserController

```

5.      Set<Role> roles= new HashSet<>();
6.      Role roleUser = new Role("ROLE_USER");//initialisation du rôle ROLE_USER
7.      roles.add(roleUser);
8.      user.setRoles(roles);
9.      user.setActive(0);
10.
11.      Set<Role> roleFromDB = extractRole_Java8(user.getRoles(),
roleService.getAllRolesStream());
12.      user.getRoles().removeAll(user.getRoles());
13.      user.setRoles(roleFromDB);
14.      User userSave = userService.saveOrUpdateUser(user);
15.      logger.info("userSave : " + userSave.toString());
16.      return new ResponseEntity<User>(user, HttpStatus.CREATED);
17.  }
```

### Et voici les méthodes utilitaires pour compléter l'extraction

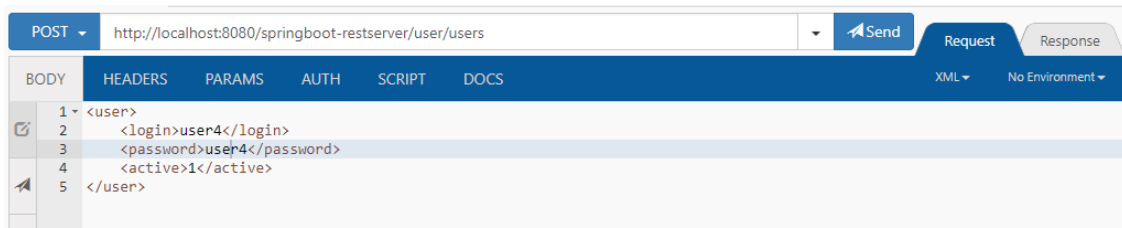
```

1.
2.      private Set<Role> extractRole_Java8(Set<Role> rolesSetFromUser, Stream<Role>
roleStreamFromDB) {
3.          // Collect UI role names
4.          Set<String> uiRoleNames = rolesSetFromUser.stream()
5.              .map(Role::getRoleName)
6.              .collect(Collectors.toCollection(HashSet::new));
7.          // Filter DB roles
8.          return roleStreamFromDB
9.              .filter(role -> uiRoleNames.contains(role.getRoleName()))
10.             .collect(Collectors.toSet());
11.     }
12.
13.     private Set<Role> extractRoleUsingCompareTo_Java8(Set<Role> rolesSetFromUser,
Stream<Role> roleStreamFromDB) {
14.         return roleStreamFromDB
15.             .filter(roleFromDB -> rolesSetFromUser.stream()
16.                 .anyMatch(roleFromUser -> roleFromUser.compareTo(roleFromDB) == 0))
17.             .collect(Collectors.toCollection(HashSet::new));
18.     }
19.
20.     private Set<Role> extractRole_BeforeJava8(Set<Role> rolesSetFromUser, Collection<Role>
rolesFromDB) {
21.         Set<Role> rolesToAdd = new HashSet<>();
22.         for(Role roleFromUser:rolesSetFromUser){
23.             for (Role roleFromDB:rolesFromDB){
24.                 if(roleFromDB.compareTo(roleFromUser)==0){
25.                     rolesToAdd.add(roleFromDB);
26.                     break;
27.                 }
28.             }
29.         }
30.         return rolesToAdd;
31.     }
```

J'ai ajouté quelques méthodes utilitaires en **Java8** et **avant Java8** juste pour mieux montrer la différence dans l'utilisation des Collections entre les versions de Java. Ces trois méthodes font strictement la même chose : récupérer les rôles utilisateurs de la base de données. Car en principe, je ne crée pas de rôles directement, mais je fais une requête dans la base de données pour récupérer le rôle à partir de **roleName** fourni par le **User**.

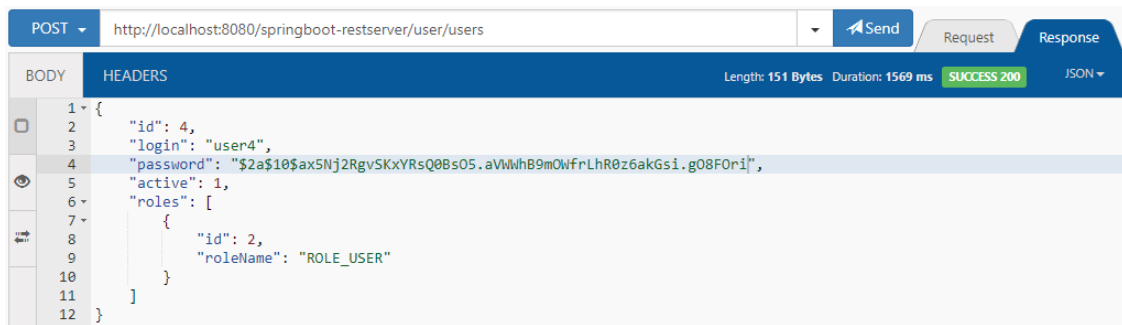
La méthode **extractRole\_Java8()** est écrite exclusivement en Java8 et utilise la classe **Stream** et les filtres pour extraire les rôles utilisateurs. La méthode **extractRoleUsingCompareTo\_Java8()** est pareille que la méthode ci-dessus, mais utilise la méthode de comparaison **compareTo()** de la classe **Role**. La méthode **extractRole\_BeforeJava8()** est la méthode classique de parcours d'une Collection.

Voici la requête de création d'un utilisateur avec une **requête POST** en utilisant le client REST **Boomerang**. Voici l'URI de création d'un utilisateur : <http://localhost:8080/springboot-restserver/user/users>.



Requête HTTP POST de création d'un utilisateur

Réponse du serveur après création d'un utilisateur



Réponse création d'un utilisateur



On constate bien dans la réponse que le mot de passe est crypté comme prévu

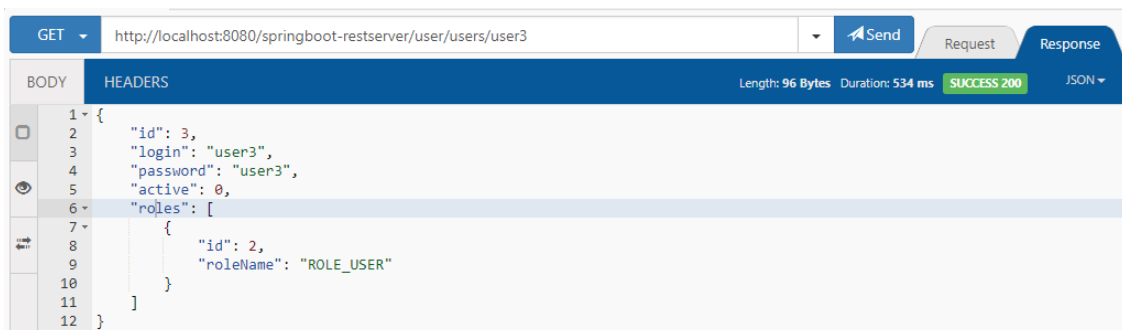
## I-B-3-f - Recherche d'un utilisateur par son login

### Service de recherche d'un utilisateur par son login

```
1.
2. @GetMapping(value = "/users/{loginName}")
3. public ResponseEntity<User> findUserByLogin(@PathVariable("loginName") String login) {
4.     User user = userService.findByLogin(login);
5.     logger.debug("Utilisateur trouvé : " + user);
6.     return new ResponseEntity<User>(user, HttpStatus.FOUND);
7. }
```

Recherchons par exemple l'utilisateur ayant le login : **user3** (requête HTTP GET) <http://localhost:8080/springboot-restserver/user/users/user3>.

On obtient:



Recherche utilisateur par login

## I-B-3-g - Service modification d'un utilisateur

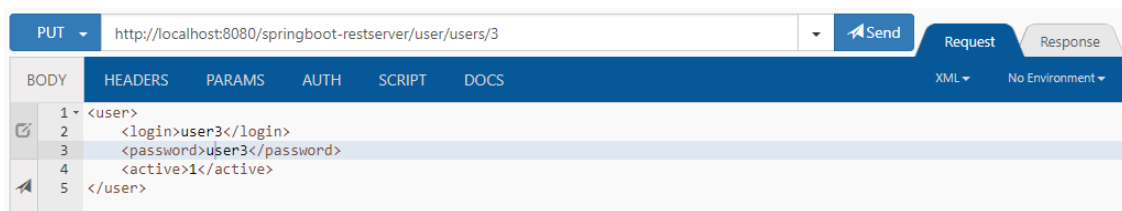
### Service qui permet de modifier un utilisateur

```

1.
2.     @PutMapping(value = "/users/{id}")
3.     public ResponseEntity<User> updateUser(@PathVariable(value = "id") Long id, @RequestBody
4.     User user) {
5.         User userToUpdate = userService.getUserById(id);
6.         if (userToUpdate == null) {
7.             logger.debug("L'utilisateur avec l'identifiant " + id + " n'existe pas");
8.             return new ResponseEntity<User>(user, HttpStatus.NOT_FOUND);
9.         }
10.
11.         logger.info("UPDATE ROLE: "+userToUpdate.getRoles().toString());
12.         userToUpdate.setLogin(user.getLogin());
13.         userToUpdate.setPassword(user.getPassword());
14.         userToUpdate.setActive(user.getActive());
15.         User userUpdated = userService.saveOrUpdateUser(userToUpdate);
16.         return new ResponseEntity<User>(userUpdated, HttpStatus.OK);
17.     }

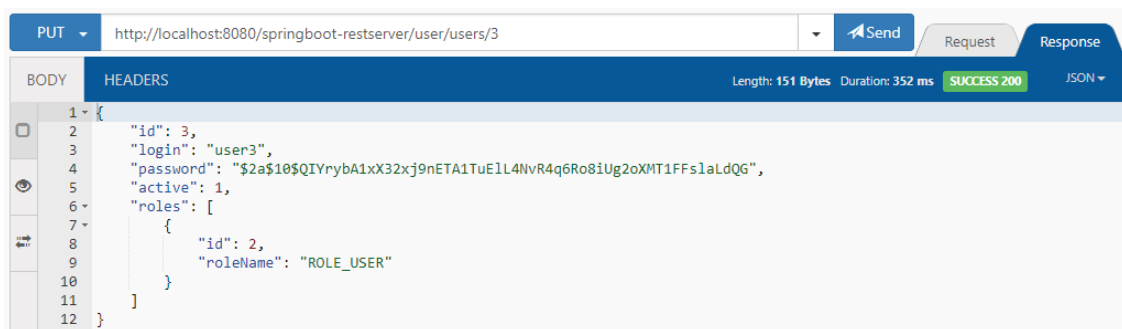
```

Je vais procéder à la modification de l'utilisateur ci-dessus ayant l'identifiant id=3 en activant par exemple son compte. Son statut passera donc de 0 à 1. Configurer donc votre client pour envoyer une **requête PUT** comme ci-dessous. Adresse de la requête : **http://localhost:8080/springboot-restserver/user/users/3**.



Requête HTTP PUT à envoyer au serveur

### Réponse à la requête HTTP PUT



Réponse à la requête HTTP PUT

## I-B-3-h - Suppression d'un utilisateur

### Méthode HTTP DELETE de suppression d'un utilisateur

```

1.
2.     @DeleteMapping(value = "/users/{id}")
3.     public ResponseEntity<Void> deleteUser(@PathVariable(value = "id") Long id) {
4.         userService.deleteUser(id);
5.         return new ResponseEntity<Void>(HttpStatus.GONE);
6.     }

```

Requête de suppression : **http://localhost:8080/springboot-restserver/user/users/3**.

## I-C - Gestion des exceptions

Dans cette section, je vais mettre en place la gestion des exceptions en utilisant l'annotation **@ControllerAdvice**.

### I-C-1 - Pourquoi utiliser @ControllerAdvice

L'annotation **@ControllerAdvice** est une spécialisation de l'annotation de **@Component** introduite depuis **Spring-3.2**. Elle permet de gérer de façon plus globale les exceptions dans un service REST. Une classe portant cette annotation est détectée automatiquement par Spring au chargement de l'application. Cette annotation prend en charge trois autres annotations très importantes dont **@ExceptionHandler**, **@InitBinder** et **@ModelAttribute**. Un contrôleur annoté par **@ControllerAdvice** utilise l'annotation **@ExceptionHandler** pour intercepter des exceptions dites globales, quelles que soient leurs origines. **@InitBinder** permet une initialisation globale et **@ModelAttribute** permet de créer un objet model global (par exemple : création et initialisation d'une vue ou d'une page de l'application).

Pour mieux comprendre, je vais créer un nouveau package nommé **exception** et une classe **GlobalHandlerControllerException** de gestion globale des exceptions annotée par **@ControllerAdvice**, et quelques-unes de ses méthodes annotées par **@ExceptionHandler**, **@InitBinder** and **@ModelAttribute**.

#### Classe global de gestion des exceptions grâce à l'annotation @ControllerAdvice

```

1.
2. package com.bnquingo.springbootrestserver.exception;
3.
4. import javax.servlet.http.HttpServletRequest;
5.
6. import org.springframework.http.HttpStatus;
7. import org.springframework.http.ResponseEntity;
8. import org.springframework.ui.Model;
9. import org.springframework.web.bind.WebDataBinder;
10. import org.springframework.web.bind.annotation.ControllerAdvice;
11. import org.springframework.web.bind.annotation.ExceptionHandler;
12. import org.springframework.web.bind.annotation.InitBinder;
13. import org.springframework.web.bind.annotation.ModelAttribute;
14. import org.springframework.web.servlet.ModelAndView;
15. import org.springframework.web.servlet.mvc.method.annotation.ResponseEntityExceptionHandler;
16.
17. @ControllerAdvice(basePackages = {"com.bnquingo.springbootrestserver"})
18. //Spring 3.2 And Above
19. public class GlobalHandlerControllerException extends ResponseEntityExceptionHandler{
20.
21.     @InitBinder
22.     public void dataBinding(WebDataBinder binder) {
23.         //Vous pouvez initialiser toute autre donnée ici
24.     }
25.
26.     @ModelAttribute //la variable "technicalError" pourra être exploité n'importe où dans l'application
27.     public void globalAttributes(Model model) {
28.         model.addAttribute("technicalError", "Une erreur technique est survenue !");
29.     }
30.
31.     @ExceptionHandler(TechnicalErrorException.class)
32.     public ModelAndView technicalErrorException(Exception exception) {
33.         ModelAndView mav = new ModelAndView();
34.         mav.addObject("exception", exception.getMessage());
35.         mav.setViewName("error");
36.         return mav;
37.     }
38.
39.     @ExceptionHandler(Exception.class) //toutes les autres erreurs non gérées sont interceptées ici
40.     public ResponseEntity<BusinessResourceExceptionResponse> unknowError(HttpServletRequest req, Exception ex) {
41.         BusinessResourceExceptionResponse response = new BusinessResourceExceptionResponse();
42.         response.setErrorCode("Technical Error");
43.         response.setErrorMessage(ex.getMessage());
44.         response.setRequestURL(req.getRequestURL().toString());

```

### Classe global de gestion des exceptions grâce à l'annotation @ControllerAdvice

```

43.         return new ResponseEntity<BusinessResourceExceptionResponse>(response,
44.             HttpStatus.INTERNAL_SERVER_ERROR);
45.     }
46.     @ExceptionHandler(BusinessResourceException.class)
47.     public
48.     ResponseEntity<BusinessResourceExceptionResponse> resourceNotFound(HttpServletRequest req,
49.         BusinessResourceException ex) {
50.         BusinessResourceExceptionResponse response = new BusinessResourceExceptionResponse();
51.         response.setStatus(ex.getStatus());
52.         response.setErrorCode(ex.getErrorCode());
53.         response.setErrorMessage(ex.getMessage());
54.         response.setRequestURL(req.getRequestURL().toString());
55.         return new ResponseEntity<BusinessResourceExceptionResponse>(response,
56.             ex.getStatus());
57.     }

```

Comme on peut le constater dans la classe ci-dessus, j'ai annoté la classe par `@ControllerAdvice`. Cette classe est capable d'intercepter et de gérer plusieurs types d'erreurs grâce à l'annotation `@ExceptionHandler` avec en paramètre le type d'exception. On peut aussi lui passer plusieurs classes d'exceptions pour le même Handler. Il faut aussi noter le passage en paramètre de l'attribut **basePackages** qui permet d'indiquer dans quels packages se trouvent les contrôleurs à prendre en compte par cette classe (ici, tous les contrôleurs du package **com.bnguimgo.springbootrestserver** sont concernés et leurs erreurs seront donc gérées par **GlobalHandlerControllerException**).

Pour les erreurs techniques non spécifiées, on renvoie vers la vue **error.jsp**.

### Toutes les exceptions non traitées seront interceptées par la méthode

```

1.
2.
3.     @ExceptionHandler(Exception.class) //toutes les autres erreurs non gérées sont interceptées ici
4.     public ResponseEntity<BusinessResourceExceptionResponse> unknowError(HttpServletRequest req, Exception ex) {
5.         BusinessResourceExceptionResponse response = new BusinessResourceExceptionResponse();
6.         response.setErrorCode("Technical Error");
7.         response.setErrorMessage(ex.getMessage());
8.         response.setRequestURL(req.getRequestURL().toString());
9.         return new ResponseEntity<BusinessResourceExceptionResponse>(response,
10.             HttpStatus.INTERNAL_SERVER_ERROR);
11.     }

```

L'attribut global nommé **technicalError** pourra être utilisé partout dans l'application.

J'ai ajouté trois classes utilitaires de gestion des exceptions. Deux classes standards de gestion des exceptions que j'ai nommées **BusinessResourceException** et **TechnicalErrorException**. Et enfin une classe POJO nommée **BusinessResourceExceptionResponse** qui correspond à l'objet qui va servir à stocker les messages d'erreurs.

### Classe d'exception classique BusinessResourceException

```

1.
2. package com.bnguimgo.springbootrestserver.exception;
3.
4. import org.springframework.http.HttpStatus;
5.
6. public class BusinessResourceException extends RuntimeException {
7.
8.     private static final long serialVersionUID = 1L;
9.     private Long resourceId;
10.    private String errorCode;
11.    private HttpStatus status;
12.
13.    public BusinessResourceException(String message) {

```



### Classe d'exception classique BusinessException

```

14.     super(message);
15. }
16.
17. public BusinessException(Long resourceId, String message) {
18.     super(message);
19.     this.resourceId = resourceId;
20. }
21. public BusinessException(Long resourceId, String errorCode, String message) {
22.     super(message);
23.     this.resourceId = resourceId;
24.     this.errorCode = errorCode;
25. }
26.
27. public BusinessException(String errorCode, String message) {
28.     super(message);
29.     this.errorCode = errorCode;
30. }
31.
32. public BusinessException(String errorCode, String message, HttpStatus status) {
33.     super(message);
34.     this.errorCode = errorCode;
35.     this.status = status;
36. }
37.
38. public Long getResourceId() {
39.     return resourceId;
40. }
41.
42. public void setResourceId(Long resourceId) {
43.     this.resourceId = resourceId;
44. }
45.
46. public String getErrorCode() {
47.     return errorCode;
48. }
49.
50. public void setErrorCode(String errorCode) {
51.     this.errorCode = errorCode;
52. }
53.
54. public HttpStatus getStatus() {
55.     return status;
56. }
57.
58. public void setStatus(HttpStatus status) {
59.     this.status = status;
60. }
61. }

```

### Classe de gestion d'erreurs techniques TechnicalErrorException

```

1.
2. package com.bnguimgo.springbootrestserver.exception;
3.
4. public class TechnicalErrorException extends RuntimeException {
5.
6.     private static final long serialVersionUID = -811807278404114373L;
7.
8.     private Long id;
9.
10.    public TechnicalErrorException() {
11.        super();
12.    }
13.
14.    public TechnicalErrorException(String message) {
15.        super(message);
16.    }
17.
18.    public TechnicalErrorException(Throwable cause) {
19.        super(cause);
20.    }

```



### Classe de gestion d'erreurs techniques TechnicalErrorException

```
21.
22.     public TechnicalErrorException(String message, Throwable throwable) {
23.         super(message, throwable);
24.     }
25.
26.     public TechnicalErrorException(Long id) {
27.         super(id.toString());
28.         this.id = id;
29.     }
30.
31.     public Long getId() {
32.         return id;
33.     }
34.     public void setId(Long id) {
35.         this.id = id;
36.     }
37. }
```

### Classe POJO BusinessExceptionResponse de persistance des messages d'erreurs ExceptionResponse

```
1.
2.
3. package com.bnguimgo.springbootrestserver.exception;
4.
5. import org.springframework.http.HttpStatus;
6.
7. public class BusinessExceptionResponse {
8.
9.     private String errorCode;
10.    private String errorMessage;
11.    private String requestURL;
12.    private HttpStatus status;
13.
14.    public BusinessExceptionResponse() {
15.    }
16.
17.    public String getErrorCode() {
18.        return errorCode;
19.    }
20.
21.    public void setErrorCode(String errorCode) {
22.        this.errorCode = errorCode;
23.    }
24.
25.    public String getErrorMessage() {
26.        return errorMessage;
27.    }
28.
29.    public void setErrorMessage(String errorMessage) {
30.        this.errorMessage = errorMessage;
31.    }
32.
33.    public void setRequestURL(String url) {
34.        this.requestURL = url;
35.    }
36.
37.    public String getRequestURL() {
38.        return requestURL;
39.    }
40.
41.    public HttpStatus getStatus() {
42.        return status;
43.    }
44.
45.    public void setStatus(HttpStatus status) {
46.        this.status = status;
47.    }
48. }
```

## I-C-2 - Test des services avec exceptions

Avec l'intégration des exceptions, il est tout à fait normal de modifier nos services précédents pour prendre en compte la gestion des exceptions.

Voici ce que devient l'interface UserService avec les exceptions

```
1.
2. package com.bnguimgo.springbootrestserver.service;
3. import java.util.Collection;
4.
5. import com.bnguimgo.springbootrestserver.exception.BusinessResourceException;
6. import com.bnguimgo.springbootrestserver.model.User;
7.
8. public interface UserService {
9.
10.     Collection<User> getAllUsers();
11.
12.     User getUserById(Long id) throws BusinessResourceException;
13.
14.     User findByLogin(String login) throws BusinessResourceException;
15.
16.     User saveOrUpdateUser(User user) throws BusinessResourceException;
17.
18.     void deleteUser(Long id) throws BusinessResourceException;
19. }
```

Mise à jour de UserServiceImpl avec gestion des exceptions

```
1.
2. package com.bnguimgo.springbootrestserver.service;
3. import java.util.Collection;
4.
5. import org.apache.commons.collections4.IteratorUtils;
6. import org.springframework.beans.factory.annotation.Autowired;
7. import org.springframework.http.HttpStatus;
8. import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
9. import org.springframework.stereotype.Service;
10. import org.springframework.transaction.annotation.Transactional;
11.
12. import com.bnguimgo.springbootrestserver.dao.UserRepository;
13. import com.bnguimgo.springbootrestserver.exception.BusinessResourceException;
14. import com.bnguimgo.springbootrestserver.exception.UserNotFoundException;
15. import com.bnguimgo.springbootrestserver.model.User;
16.
17. @Service(value = "userService")
18. public class UserServiceImpl implements UserService {
19.
20.     @Autowired
21.     private UserRepository userRepository;
22.
23.     @Autowired
24.     private BCryptPasswordEncoder bCryptPasswordEncoder;
25.
26.     @Override
27.     public User findByLogin(String login) throws BusinessResourceException {
28.         User userFound = userRepository.findByLogin(login);
29.         return userFound;
30.     }
31.
32.     @Override
33.     public Collection<User> getAllUsers() {
34.         return IteratorUtils.toList(userRepository.findAll().iterator());
35.     }
36.
37.     @Override
38.     public User getUserById(Long id) throws BusinessResourceException{
39.         return userRepository.findOne(id);
40.     }
41. }
```

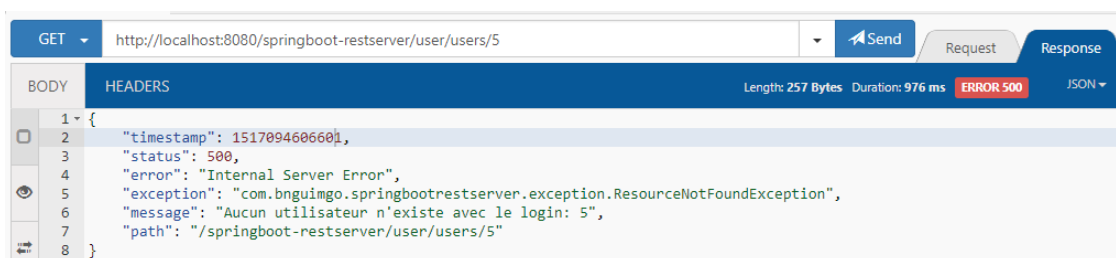
## Mise à jour de UserServiceImpl avec gestion des exceptions

```

42.     @Override
43.     @Transactional(readOnly=false)
44.     public User saveOrUpdateUser(User user) {
45.         try{
46.             user.setPassword(bCryptPasswordEncoder.encode(user.getPassword()));
47.             return userRepository.save(user);
48.         } catch (Exception ex) {
49.
50.             throw new BusinessException("Create Or Update User Error", "Erreur de création ou de mise à jour de l'u
51.             HttpStatus.INTERNAL_SERVER_ERROR);
52.         }
53.     @Override
54.     @Transactional(readOnly=false)
55.     public void deleteUser(Long id) throws UserNotFoundException {
56.         try{
57.             userRepository.delete(id);
58.         } catch (Exception ex) {
59.
60.             throw new BusinessException("Delete User Error", "Erreur de suppression de l'utilisateur avec l'identif
61.             HttpStatus.INTERNAL_SERVER_ERROR);
62.         }
63.     }

```

Pour mettre en évidence l'intérêt de l'annotation de `@ControllerAdvice`, je vais faire un premier cas de test en cherchant un utilisateur qui n'existe pas et avec l'annotation `@ControllerAdvice` désactivée (*mettre en commentaire*). Résultat:



### Exception sans l'annotation `@ControllerAdvice`

On obtient **ERROR 500** et un ensemble d'informations pas très personnalisées.

Et voici l'exception renvoyée lorsque l'annotation `@ControllerAdvice` est activée



### Exception lorsque `@ControllerAdvice` est activée

On obtient **INVALID 404**, mais avec une suite d'erreurs plus claire et exploitable. Par exemple : **un code d'erreur, le message, et une URL** bien précise et complète.

## I-D - Tests unitaires des couches

Chaque couche de l'application peut être testée unitairement. Et comme vous pouvez le constater, chaque couche a une dépendance vis-à-vis de la couche immédiatement inférieure. Dans le cas des tests unitaires, l'idée est de s'affranchir de cette dépendance et de tester unitairement chaque méthode de la couche. C'est ce qui explique le plus souvent l'usage des **Mocks Objects** (*Objets mocks = Objets factices ou objets sans existence réelle*) dans les tests unitaires (ce qui n'est pas le cas dans les tests d'intégration).

Il faut donc trouver quel Mock Object est adapté pour chaque couche de l'application, car ça change en fonction des besoins et des cas de tests.

Mais, avant de commencer avec les tests, j'ai ajouté quelques dépendances nécessaires pour les tests unitaires et les tests d'intégration. Voici les deux dépendances :

### Dépendances nécessaires pour les tests

```

1.
2.     <dependency>
3.         <groupId>com.jayway.jsonpath</groupId>
4.         <artifactId>json-path</artifactId>
5.         <scope>test</scope>
6.     </dependency>
7.     <dependency>
8.         <groupId>org.hsqldb</groupId>
9.         <artifactId>hsqldb</artifactId>
10.        <scope>test</scope>
11.    </dependency>

```

Et voici le pom.xml complet de l'application:

### pom.xml complet de l'application et des tests

```

1.
2. <?xml version="1.0" encoding="UTF-8"?>
3. <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance"
4.     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
maven-4.0.0.xsd">
5.     <modelVersion>4.0.0</modelVersion>
6.
7.     <groupId>com.bnnguimgo</groupId>
8.     <artifactId>springboot-restserver</artifactId>
9.     <version>0.0.1-SNAPSHOT</version>
10.    <packaging>war</packaging>
11.
12.    <name>springboot-restserver</name>
13.    <description>Demo project for Spring Boot</description>
14.
15.    <parent>
16.        <groupId>org.springframework.boot</groupId>
17.        <artifactId>spring-boot-starter-parent</artifactId>
18.        <version>1.5.9.RELEASE</version>
19.        <relativePath/> <!-- lookup parent from repository -->
20.    </parent>
21.
22.    <properties>
23.        <start-
class>com.bnnguimgo.restclient.com.bnnguimgo.springbootrestserver.SpringbootRestserverApplication</start-
class>
24.        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
25.        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
26.        <java.version>1.8</java.version>
27.        <tomcat.version>8.0.41</tomcat.version><!-- This override default embedded Tomcat --
>
28.        <maven.test.skip>true</maven.test.skip>
29.    </properties>
30.
31.    <dependencies>

```

### pom.xml complet de l'application et des tests

```

32.         <dependency>
33.             <groupId>org.springframework.boot</groupId>
34.             <artifactId>spring-boot-starter-data-jpa</artifactId>
35.         </dependency>
36.         <dependency>
37.             <groupId>org.springframework.boot</groupId>
38.             <artifactId>spring-boot-starter-web</artifactId>
39.         </dependency>
40.         <!-- Spring Security: juste pour encrypter le mot de passe -->
41.         <dependency>
42.             <groupId>org.springframework.boot</groupId>
43.             <artifactId>spring-boot-starter-security</artifactId>
44.         </dependency>
45.
46.         <!-- Permet l'utilisation des collections Apache -->
47.         <dependency>
48.             <groupId>org.apache.commons</groupId>
49.             <artifactId>commons-collections4</artifactId>
50.             <version>4.0</version>
51.         </dependency>
52.
53.         <!-- Base de données utilisée par l'application -->
54.         <dependency>
55.             <groupId>com.h2database</groupId>
56.             <artifactId>h2</artifactId>
57.             <scope>runtime</scope>
58.         </dependency>
59.
60.         <!-- Dépendances pour les tests uniquement -->
61.         <dependency>
62.             <groupId>org.springframework.boot</groupId>
63.             <artifactId>spring-boot-starter-test</artifactId>
64.         </dependency>
65.         <dependency>
66.             <groupId>com.jayway.jsonpath</groupId>
67.             <artifactId>json-path</artifactId>
68.             <scope>test</scope>
69.         </dependency>
70.         <!-- Base de données pour les tests -->
71.         <dependency>
72.             <groupId>org.hsqldb</groupId>
73.             <artifactId>hsqldb</artifactId>
74.             <scope>test</scope>
75.         </dependency>
76.     </dependencies>
77.
78.     <build>
79.         <finalName>springboot-restserver</finalName>
80.         <plugins>
81.             <plugin>
82.                 <groupId>org.springframework.boot</groupId>
83.                 <artifactId>spring-boot-maven-plugin</artifactId>
84.             </plugin>
85.         </plugins>
86.     </build>
87.
88. </project>

```

## I-D-1 - Tests unitaires couche DAO

La couche **DAO** est la couche la plus basse de l'application et, de ce fait, elle communique le plus souvent avec la base de données. Pour la tester, on ne cherche pas à savoir comment se passe l'extraction des données, mais on vérifie plutôt que la couche DAO est capable de fournir les données dont on a besoin : d'où l'utilisation d'un **Mock Object**. Pour répondre à ce défi, Spring Boot a mis à notre disposition la classe **TestEntityManager** qui est l'équivalent Mock de **JPA EntityManager**.

### Voici le service d'accès aux données utilisateur à tester

```
1.
2. public interface UserRepository extends JpaRepository<User, Long> {
3.     User findByLogin(String login);
4. }
```

### Et ci-dessous l'initialisation des tests

```
1.
2. @RunWith(SpringRunner.class) //permet d'établir une liaison entre JUnit et Spring
3. @DataJpaTest
4. public class UserRepositoryTest {
5.
6.     @Autowired
7.     private EntityManager entityManager;
8.
9.     @Autowired
10.    private UserRepository userRepository;
11.
12.    User user = new User("Dupont", "password", 1);
13. }
```

**@RunWith(SpringRunner.class)** permet d'établir la liaison l'implémentation Spring de JUnit, donc c'est tout naturel qu'on l'utilise pour un test unitaire.

**@DataJpaTest** est une implémentation Spring de JPA qui fournit une configuration intégrée de la base de données **H2, Hibernate, SpringData, et la DataSource**. Cette annotation active également la détection des entités annotées par **Entity**, et intègre aussi la gestion des **logs SQL**.

Notre cas de test consiste à **tester** la recherche d'un utilisateur par son **login**. Pour ce faire, on doit préalablement enregistrer un utilisateur dans le **setup**.

### Test unitaire de la recherche d'un utilisateur par son login

```
1.
2. package com.bnguimgo.springbootrestserver.dao;
3.
4. import org.junit.Before;
5. import org.junit.Test;
6. import org.junit.runner.RunWith;
7. import static org.junit.Assert.*;
8.
9. import java.util.List;
10.
11. import static org.hamcrest.CoreMatchers.*;
12. import org.springframework.beans.factory.annotation.Autowired;
13. import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
14. import org.springframework.boot.test.autoconfigure.orm.jpa.TestEntityManager;
15. import org.springframework.test.context.junit4.SpringRunner;
16.
17. import com.bnguimgo.springbootrestserver.model.User;
18.
19. @RunWith(SpringRunner.class)
20. @DataJpaTest
21. public class UserRepositoryTest {
22.
23.     @Autowired
24.     private TestEntityManager entityManager;
25.     @Autowired
26.     private UserRepository userRepository;
27.     User user = new User("Dupont", "password", 1);
28.
29.     @Before
30.     public void setup() {
31.         entityManager.persist(user); //on sauvegarde l'objet user au début de chaque test
32.         entityManager.flush();
33.     }
34.     @Test
35.     public void testFindAllUsers() {
36.         List<User> users = userRepository.findAll();
```

### Test unitaire de la recherche d'un utilisateur par son login

```

37.
    assertThat(4, is(users.size())); //on a trois users dans le fichier d'initialisation data.sql et un utilisateur
38.    }
39.
40.    @Test
41.    public void testSaveUser() {
42.        User user = new User("Paul", "password", 1);
43.        User userSaved = userRepository.save(user);
44.        assertNotNull(userSaved.getId());
45.        assertThat("Paul", is(userSaved.getLogin()));
46.    }
47.    @Test
48.    public void testFindByLogin() {
49.        User userFromDB = userRepository.findByLogin("user2");
50.
51.        assertThat("user2", is(userFromDB.getLogin())); //user2 a été créé lors de l'initialisation du fichier data.sql
52.    }
53.
54.    @Test
55.    public void testDeleteUser() {
56.        userRepository.delete(user.getId());
57.        User userFromDB = userRepository.findByLogin(user.getLogin());
58.        assertNull(userFromDB);
59.    }
60.
61.    @Test
62.    public void testUpdateUser() { //Test si le compte utilisateur est désactivé
63.        User userToUpdate = userRepository.findByLogin(user.getLogin());
64.        userToUpdate.setActive(0);
65.        userRepository.save(userToUpdate);
66.        User userUpdatedFromDB = userRepository.findByLogin(userToUpdate.getLogin());
67.        assertNotNull(userUpdatedFromDB);
68.        assertThat(0, is(userUpdatedFromDB.getActive()));
69.    }

```

**@TestEntityManager** permet de persister les données en base lors des tests. Toutes les méthodes ont été testées.



J'ai ajouté le framework **hamcrest** (fourni par Spring Boot) qui apporte plus de flexibilité pour l'utilisation de la méthode de test **assertThat()**.

## I-D-2 - Tests unitaires couche service

La couche de service a une dépendance directe avec la couche DAO. Cependant, on n'a pas besoin de savoir comment se passe la persistance au niveau de la couche DAO. Le plus important c'est que cette couche renvoie les données sollicitées. On va donc mocker cette couche en utilisant le **framework Mockito**

Voici le service à tester sans le Mock:

```

1.
2. package com.bnguimgo.springbootrestserver.service;
3. import java.util.Collection;
4.
5. import org.apache.commons.collections4.IteratorUtils;
6. import org.springframework.beans.factory.annotation.Autowired;
7. import org.springframework.http.HttpStatus;
8. import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
9. import org.springframework.stereotype.Service;
10. import org.springframework.transaction.annotation.Transactional;
11.
12. import com.bnguimgo.springbootrestserver.dao.UserRepository;
13. import com.bnguimgo.springbootrestserver.exception.BusinessResourceException;
14. import com.bnguimgo.springbootrestserver.exception.UserNotFoundException;
15. import com.bnguimgo.springbootrestserver.model.User;
16.

```

### Voici le service à tester sans le Mock:

```

17. @Service(value = "userService")
18. public class UserServiceImpl implements UserService {
19.
20.     @Autowired
21.     private UserRepository userRepository;
22.
23.     @Autowired
24.     private BCryptPasswordEncoder bCryptPasswordEncoder;
25.
26.     @Override
27.     public User findByLogin(String login) throws BusinessResourceException {
28.         User userFound = userRepository.findByLogin(login);
29.         return userFound;
30.     }
31.
32.     @Override
33.     public Collection<User> getAllUsers() {
34.         return IteratorUtils.toList(userRepository.findAll().iterator());
35.     }
36.
37.     @Override
38.     public User getUserById(Long id) throws BusinessResourceException{
39.         return userRepository.findOne(id);
40.     }
41.
42.     @Override
43.     @Transactional(readOnly=false)
44.     public User saveOrUpdateUser(User user) {
45.         try{
46.             user.setPassword(bCryptPasswordEncoder.encode(user.getPassword()));
47.             return userRepository.save(user);
48.         }catch(Exception ex){
49.
50.             throw new BusinessResourceException("Create Or Update User Error", "Erreur de création ou de mise à jour de l'u
51.             HttpStatus.INTERNAL_SERVER_ERROR);
52.         }
53.
54.     @Override
55.     @Transactional(readOnly=false)
56.     public void deleteUser(Long id) throws UserNotFoundException {
57.         try{
58.             userRepository.delete(id);
59.         }catch(Exception ex){
60.
61.             throw new BusinessResourceException("Delete User Error", "Erreur de suppression de l'utilisateur avec l'identif
62.             HttpStatus.INTERNAL_SERVER_ERROR);
63.         }
64.     }
65. }

```



*Pour éviter d'avoir des problèmes de configuration des tests, il faut créer exactement les mêmes packages dans src/test/java. Par exemple, pour tester le service UserService, créer le package : **com.nguimgo.springbootrestserver.service***

### Voici la classe de test correspondante: recherche d'un utilisateur par son login

```

1.
2. package com.nguimgo.springbootrestserver.service;
3.
4. import static org.hamcrest.CoreMatchers.is;
5. import static org.junit.Assert.assertEquals;
6. import static org.junit.Assert.assertNotNull;
7. import static org.junit.Assert.assertThat;
8. import static org.mockito.Matchers.any;
9. import static org.mockito.Mockito.verify;
10.

```



## Voici la classe de test correspondante: recherche d'un utilisateur par son login

```
11. import java.util.Arrays;
12. import java.util.Collection;
13. import java.util.HashSet;
14. import java.util.List;
15. import java.util.Set;
16.
17. import org.junit.Test;
18. import org.junit.runner.RunWith;
19. import org.mockito.Mockito;
20. import org.springframework.beans.factory.annotation.Autowired;
21. import org.springframework.boot.test.context.TestConfiguration;
22. import org.springframework.boot.test.mock.mockito.MockBean;
23. import org.springframework.context.annotation.Bean;
24. import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
25. import org.springframework.test.context.junit4.SpringRunner;
26.
27. import com.bnguimgo.springbootrestserver.dao.UserRepository;
28. import com.bnguimgo.springbootrestserver.model.Role;
29. import com.bnguimgo.springbootrestserver.model.User;
30.
31. @RunWith(SpringRunner.class)
32. public class UserServiceImplTest {
33.
34.     @TestConfiguration //création des beans nécessaires pour les tests
35.     static class UserServiceImplTestContextConfiguration {
36.
37.         @Bean //bean de service
38.         public UserService userService () {
39.             return new UserServiceImpl();
40.         }
41.
42.         @Bean //nécessaire pour hacher le mot de passe sinon échec des tests
43.         public BCryptPasswordEncoder passwordEncoder () {
44.             BCryptPasswordEncoder bCryptPasswordEncoder = new BCryptPasswordEncoder();
45.             return bCryptPasswordEncoder;
46.         }
47.     }
48.
49.     @Autowired
50.     private UserService userService;
51.
52.     @MockBean //création d'un mockBean pour UserRepository
53.     private UserRepository userRepository;
54.
55.     User user = new User("Dupont", "password", 1);
56.
57.     @Test
58.     public void testFindAllUsers() throws Exception {
59.         User user = new User("Dupont", "password", 1);
60.         Role role = new Role("USER_ROLE"); //initialisation du role utilisateur
61.         Set<Role> roles = new HashSet<>();
62.         roles.add(role);
63.         user.setRoles(roles);
64.         List<User> allUsers = Arrays.asList(user);
65.         Mockito.when(userRepository.findAll()).thenReturn(allUsers);
66.         Collection<User> users = userService.getAllUsers();
67.         assertNotNull(users);
68.         assertEquals(users, allUsers);
69.         assertEquals(users.size(), allUsers.size());
70.         verify(userRepository).findAll();
71.     }
72.
73.     @Test
74.     public void testSaveUser() throws Exception {
75.         User user = new User("Dupont", "password", 1);
76.         User userMock = new User(1L, "Dupont", "password", 1);
77.         Mockito.when(userRepository.save((user))).thenReturn(userMock);
78.         User userSaved = userService.saveOrUpdateUser(user);
79.         assertNotNull(userSaved);
80.         assertEquals(userMock.getId(), userSaved.getId());
81.         assertEquals(userMock.getLogin(), userSaved.getLogin());
```

Voici la classe de test correspondante: recherche d'un utilisateur par son login

```

82.         verify(userRepository).save(any(User.class));
83.     }
84.
85.     @Test
86.     public void testFindUserByLogin() {
87.         User user = new User("Dupont", "password", 1);
88.         Mockito.when(userRepository.findByLogin(user.getLogin())).thenReturn(user);
89.         User userFromDB = userService.findByLogin(user.getLogin());
90.         assertNotNull(userFromDB);
91.         assertEquals(userFromDB.getLogin(), user.getLogin());
92.         verify(userRepository).findByLogin(any(String.class));
93.     }
94.
95.     @Test
96.     public void testDelete() throws Exception {
97.         User user = new User("Dupont", "password", 1);
98.         User userMock = new User(1L, "Dupont", "password", 1);
99.         Mockito.when(userRepository.save((user))).thenReturn(userMock);
100.        User userSaved = userService.saveOrUpdateUser(user);
101.        assertNotNull(userSaved);
102.        assertEquals(userMock.getId(), userSaved.getId());
103.        userService.deleteUser(userSaved.getId());
104.        verify(userRepository).delete(any(Long.class));
105.    }
106.
107.    @Test
108.    public void testUpdateUser() throws Exception {
109.        User userToUpdate = new User(1L, "Dupont", "password", 1);
110.        User userUpdated = new User(1L, "Paul", "password", 1);
111.        Mockito.when(userRepository.save((userToUpdate))).thenReturn(userUpdated);
112.        User userFromDB = userService.saveOrUpdateUser(userToUpdate);
113.        assertNotNull(userFromDB);
114.        assertEquals(userUpdated.getLogin(), userFromDB.getLogin());
115.        verify(userRepository).save(any(User.class));
116.    }
117. }
  
```

Pour créer un **Mock Object** DAO, Spring Boot Test fournit l'annotation **@MockBean**. Et pour exécuter le service, nous avons besoin d'un objet instance de service. On peut obtenir cette instance à travers une déclaration de classe statique interne en utilisant l'annotation **@TestConfiguration** et en annotant cette instance par **@Bean**.

L'annotation **@TestConfiguration** a pour effet de n'exposer le bean de service que lors de la phase de tests, ce qui évite sûrement les conflits. Cette annotation joue le rôle de l'annotation **@Autowired**. Le framework Mockito a été utilisé pour moquer tout le service DAO, ce qui a permis d'initialiser les tests.

En développant les services, nous avons encrypté le mot de passe pour ne pas le laisser transiter en clair dans le réseau. Il faut donc ajouter un bean **BcryptPasswordEncoder**.

Je vous laisse le soin de tester les autres méthodes de la couche de service, et aussi le test des exceptions.

### I-D-3 - Tests unitaires du contrôleur

Le contrôleur à tester est le suivant :

```

1.
2. @Controller
3. @RequestMapping("/user/*")
4. public class UserController {
5.
6.     private static final Logger logger = LoggerFactory.getLogger(UserController.class);
7.
8.     @Autowired
9.     private UserService userService;
10.    @Autowired
11.    private RoleService roleService;
  
```

### Le contrôleur à tester est le suivant :

```

12.
13.     @GetMapping(value = "/users")
14.     public ResponseEntity<Collection<User>> getAllUsers() {
15.         Collection<User> users = userService.getAllUsers();
16.         logger.info("liste des utilisateurs : " + users.toString());
17.         return new ResponseEntity<Collection<User>>(users, HttpStatus.FOUND);
18.     }
19.
20.     @PostMapping(value = "/users")
21.     @Transactional
22.     public ResponseEntity<User> saveUser(@RequestBody User user) {
23.
24.         User userExist = userService.findByLogin(user.getLogin());
25.         if (userExist != null) {
26.             logger.debug("L'utilisateur avec le login " + user.getLogin() + " existe déjà");
27.
28.             throw new BusinessException("Duplicate Login", "Erreur de création ou de mise à jour de l'utilisateur");
29.         }
30.
31.         Set<Role> roles = new HashSet<>();
32.         Role roleUser = new Role("ROLE_USER");//initialisation du rôle ROLE_USER
33.         roles.add(roleUser);
34.         user.setRoles(roles);
35.         user.setActive(0);
36.
37.         Set<Role> roleFromDB = extractRole_Java8(user.getRoles(),
38.             roleService.getAllRolesStream());
39.         user.getRoles().removeAll(user.getRoles());
40.         user.setRoles(roleFromDB);
41.         User userSave = userService.saveOrUpdateUser(user);
42.         logger.info("userSave : " + userSave.toString());
43.         return new ResponseEntity<User>(user, HttpStatus.CREATED);
44.     }
45.
46.     @GetMapping(value = "/users/{loginName}")
47.     public ResponseEntity<User> findUserByLogin(@PathVariable("loginName") String login) {
48.         User user = userService.findByLogin(login);
49.         logger.debug("Utilisateur trouvé : " + user);
50.         return new ResponseEntity<User>(user, HttpStatus.FOUND);
51.     }
52.
53.     @PutMapping(value = "/users/{id}")
54.     public ResponseEntity<User> updateUser(@PathVariable(value = "id") Long id, @RequestBody
55.         User user) {
56.
57.         User userToUpdate = userService.getUserById(id);
58.         if (userToUpdate == null) {
59.             logger.debug("L'utilisateur avec l'identifiant " + id + " n'existe pas");
60.             return new ResponseEntity<User>(user, HttpStatus.NOT_FOUND);
61.         }
62.
63.         logger.info("UPDATE ROLE: "+userToUpdate.getRoles().toString());
64.         userToUpdate.setLogin(user.getLogin());
65.         userToUpdate.setPassword(user.getPassword());
66.         userToUpdate.setActive(user.getActive());
67.         User userUpdated = userService.saveOrUpdateUser(userToUpdate);
68.         return new ResponseEntity<User>(userUpdated, HttpStatus.OK);
69.     }
70.
71.     @DeleteMapping(value = "/users/{id}")
72.     public ResponseEntity<Void> deleteUser(@PathVariable(value = "id") Long id) throws
73.         BusinessException {
74.
75.         User user = userService.getUserById(id);
76.         if (user == null) {
77.             logger.debug("L'utilisateur avec l'identifiant " + id + " n'existe pas");
78.
79.             throw new BusinessException("User Not Found", "Aucun utilisateur n'existe avec l'identifiant: "+id ,HttpStatus.FOUND);
80.         }
81.
82.         userService.deleteUser(id);

```

Le contrôleur à tester est le suivant :

```

78.         logger.debug("L'utilisateur avec l'identifiant " + id + " supprimé");
79.         return new ResponseEntity<Void>(HttpStatus.GONE);
80.     }
81.
82.     private Set<Role> extractRole_Java8(Set<Role> rolesSetFromUser, Stream<Role>
roleStreamFromDB) {
83.         // Collect UI role names
84.         Set<String> uiRoleNames = rolesSetFromUser.stream()
85.             .map(Role::getRoleName)
86.             .collect(Collectors.toCollection(HashSet::new));
87.         // Filter DB roles
88.         return roleStreamFromDB
89.             .filter(role -> uiRoleNames.contains(role.getRoleName()))
90.             .collect(Collectors.toSet());
91.     }
92.
93.     //Méthodes utilitaires
94.     private Set<Role> extractRoleUsingCompareTo_Java8(Set<Role> rolesSetFromUser,
Stream<Role> roleStreamFromDB) {
95.         return roleStreamFromDB
96.             .filter(roleFromDB -> rolesSetFromUser.stream()
97.                 .anyMatch(roleFromUser -> roleFromUser.compareTo(roleFromDB) == 0))
98.             .collect(Collectors.toCollection(HashSet::new));
99.     }
100.
101.     private Set<Role> extractRole_BeforeJava8(Set<Role> rolesSetFromUser, Collection<Role>
rolesFromDB) {
102.         Set<Role> rolesToAdd = new HashSet<>();
103.         for (Role roleFromUser:rolesSetFromUser) {
104.             for (Role roleFromDB:rolesFromDB) {
105.                 if (roleFromDB.compareTo(roleFromUser)==0) {
106.                     rolesToAdd.add(roleFromDB);
107.                     break;
108.                 }
109.             }
110.         }
111.         return rolesToAdd;
112.     }
113. }

```

Ce contrôleur dépend de la couche de service. Il ne connaît pas la couche DAO. Il faut donc créer un Mock Object de la couche de service grâce à l'annotation déjà vue précédemment **@MockBean**. Et pour indiquer le contrôleur à tester, il faut utiliser l'annotation **@WebMvcTest** en lui passant en paramètre la classe à tester. Cette annotation va aussi apporter toutes les dépendances SpringMVC nécessaires pour le cas de test.

Grâce à l'annotation **@WebMvcTest**, on peut injecter un **MockMvc** qui servira à construire des URL pour générer des requêtes HTTP. Nous allons tester la méthode du contrôleur qui recherche tous les utilisateurs de la base de données.

Créer le package de tests `com.bnguimgo.springbootrestserver.controller` et y ajouter le test `UserControllerTest`

```

1.
2. package com.bnguimgo.springbootrestserver.controller;
3.
4. import static org.hamcrest.CoreMatchers.is;
5. import static org.hamcrest.Matchers.hasSize;
6. import static org.junit.Assert.assertEquals;
7. import static org.mockito.BDDMockito.given;
8. import static org.mockito.Matchers.any;
9. import static org.mockito.Mockito.verify;
10. import static org.mockito.Mockito.when;
11. //pour les méthodes HTTP
12. import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
13. //pour JSON
14. import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.jsonPath;
15. //pour HTTP status
16. import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
17.

```

## Créer le package de tests com.bnguimbo.springbootrestserver.controller et y ajouter le test UserControllerTest

```

18. import java.util.Arrays;
19. import java.util.HashSet;
20. import java.util.List;
21. import java.util.Set;
22.
23. import org.junit.Before;
24. import org.junit.Test;
25. import org.junit.runner.RunWith;
26. import org.springframework.beans.factory.annotation.Autowired;
27. import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
28. import org.springframework.boot.test.mock.mockito.MockBean;
29. import org.springframework.http.HttpStatus;
30. import org.springframework.http.MediaType;
31. import org.springframework.test.context.junit4.SpringRunner;
32. import org.springframework.test.web.servlet.MockMvc;
33. import org.springframework.test.web.servlet.MvcResult;
34. import org.springframework.test.web.servlet.request.MockMvcRequestBuilders;
35.
36. import com.bnguimbo.springbootrestserver.model.Role;
37. import com.bnguimbo.springbootrestserver.model.User;
38. import com.bnguimbo.springbootrestserver.service.RoleService;
39. import com.bnguimbo.springbootrestserver.service.UserService;
40.
41. @RunWith(SpringRunner.class)
42. @WebMvcTest(UserController.class)
43. public class UserControllerTest {
44.
45.     @Autowired
46.     private MockMvc mockMvc;
47.
48.     @MockBean
49.     private UserService userService;
50.     @MockBean
51.     private RoleService roleService;
52.
53.     User user = new User(1L, "Dupont", "password", 1);
54.     @Before
55.     public void setUp() {
56.         //Initialisation du setup avant chaque test
57.         Role role = new Role("USER_ROLE"); //initialisation du rôle utilisateur
58.         Set<Role> roles = new HashSet<>();
59.         roles.add(role);
60.         user.setRoles(roles);
61.         List<User> allUsers = Arrays.asList(user);
62.
63.         // Mock de la couche de service
64.         given(userService.getAllUsers()).willReturn(allUsers);
65.         when(userService.getUserById(any(Long.class))).thenReturn(user);
66.         when(userService.saveOrUpdateUser(any(User.class))).thenReturn(user);
67.         when(roleService.getAllRolesStream()).thenReturn(roles.stream());
68.
69.     }
70.
71.     @Test
72.     public void testFindAllUsers() throws Exception {
73.
74.         MvcResult result = mockMvc.perform(get("/user/users")
75.             .contentType(MediaType.APPLICATION_JSON))
76.             .andExpect(status().isFound()) //statut HTTP de la réponse
77.             .andExpect(jsonPath("$", hasSize(1)))
78.             .andExpect(jsonPath("$.login", is(user.getLogin())))
79.             .andExpect(jsonPath("$.password", is(user.getPassword())))
80.             .andExpect(jsonPath("$.active", is(user.getActive())))
81.             .andReturn();
82.
83.         // ceci est une redondance, car déjà vérifié par: isFound()
84.         assertEquals("Réponse incorrecte", HttpStatus.FOUND.value(),
85.             result.getResponse().getStatus());
86.
87.         //on s'assure que la méthode de service getAllUsers() a bien été appelée

```

## Créer le package de tests com.bnguimgo.springbootrestserver.controller et y ajouter le test UserControllerTest

```

87.         verify(userService).getAllUsers();
88.     }
89.
90.     @Test
91.     public void testSaveUser() throws Exception {
92.
93.         given(userService.findByLogin("Dupont")).willReturn(null);
94.         //on exécute la requête
95.         mockMvc.perform(MockMvcRequestBuilders.post("/user/users")
96.             .contentType(MediaType.APPLICATION_XML)
97.             .accept(MediaType.APPLICATION_XML)
98.             .content("<user><login>Dupont</login><password>password</password><active>1</active></user>"))
99.             .andExpect(status().isCreated());
100.
101.         //on s'assure que la méthode de service saveOrUpdateUser(User) a bien été appelée
102.         verify(userService).saveOrUpdateUser(any(User.class));
103.
104.     }
105.
106.     @Test
107.     public void testFindUserByLogin() throws Exception {
108.         given(userService.findByLogin("Dupont")).willReturn(user);
109.         //on exécute la requête
110.         mockMvc.perform(get("/user/users/{loginName}", new String("Dupont"))
111.             .contentType(MediaType.APPLICATION_JSON)
112.             .andExpect(status().isFound())
113.             .andExpect(jsonPath("$.login", is(user.getLogin())))
114.             .andExpect(jsonPath("$.password", is(user.getPassword())))
115.             .andExpect(jsonPath("$.active", is(user.getActive()))));
116.
117.         //Résultat: on s'assure que la méthode de service findByLogin(login) a bien été appelée
118.         verify(userService).findByLogin(any(String.class));
119.     }
120.
121.     @Test
122.     public void testDeleteUser() throws Exception {
123.         // on exécute le test
124.         mockMvc.perform(MockMvcRequestBuilders.delete("/user/users/{id}", new Long(1)))
125.             .andExpect(status().isGone());
126.
127.         // On vérifie que la méthode de service deleteUser(Id) a bien été appelée
128.         verify(userService).deleteUser(any(Long.class));
129.     }
130.
131.     @Test
132.     public void testUpdateUser() throws Exception {
133.
134.         //on exécute la requête
135.         mockMvc.perform(MockMvcRequestBuilders.put("/user/users/{id}", new Long(1))
136.             .contentType(MediaType.APPLICATION_XML)
137.             .accept(MediaType.APPLICATION_XML)
138.             .content("<user><active>0</active></user>"))
139.             .andExpect(status().isOk());
140.
141.         //on s'assure que la méthode de service saveOrUpdateUser(User) a bien été appelée
142.         verify(userService).saveOrUpdateUser(any(User.class));
143.
144.     }
145. }

```

Il faut remarquer l'injection du bean **RoleService** qui est nécessaire, car un utilisateur peut avoir un ou plusieurs rôles.

## I-D-4 - Résumé sur les tests unitaires

Ce chapitre sur les tests unitaires a permis de voir que les besoins en matière de tests unitaires diffèrent d'une couche à l'autre.

### Couche DAO :

- **@DataJpaTest** pour configurer la base de données et la **Datasource** ;
- **TestEntityManager** qui est l'équivalent **Mock de JPA EntityManager**.

### Couche de service :

- **@TestConfiguration** permet de récupérer une implémentation du bean de service ;
- **@MockBean** pour créer un objet Mock de la couche DAO ;
- **Mockito** pour créer un Mock de la couche DAO.

### Couche contrôleur :

- **@WebMvcTest** permet d'indiquer le contrôleur à tester ;
- **@MockMvc** pour injecter un mock MVC qui sert à construire des requêtes HTTP ;
- **@MockBean** pour créer un mock de la couche de service.



*Il faut bien remarquer que sur toutes les couches de l'application, l'annotation **@RunWith(SpringRunner.class)** est partout présente sur la classe de test.*

## I-E - Tests d'intégration services REST

Les tests d'intégration consistent à tester une fonctionnalité ou un service complet à travers toutes les couches de l'application. Cela suppose qu'il faut penser à un déploiement complet de l'application, d'où la nécessité d'un serveur d'application, d'une base de données et, par conséquent, **pas de Mocks Objects**. En pratique, pour tester toutes les fonctionnalités d'une application, ça prend beaucoup de temps. C'est pourquoi ces tests sont séparés des tests unitaires et sont généralement programmés à des heures d'activités réduites.

Avant d'écrire le test, voici le paramétrage nécessaire à l'initialisation de la classe de test :

### Paramétrage des tests d'intégration

```

1.
2. @RunWith(SpringRunner.class)
3. @SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
4. @TestPropertySource(locations = "classpath:test.properties")
5. public class UserControllerIntegrationTest {
6.
7.     private static final Logger logger =
8.         LoggerFactory.getLogger(UserControllerIntegrationTest.class);
9.
10.    @Autowired
11.    private TestRestTemplate restTemplate;
12.
13.    @LocalServerPort //permet d'utiliser le port local du serveur, sinon une erreur "Connection refused"
14.    private int port;
15.
16.    private static final String URL= "http://
17.    localhost:"; //url du serveur REST. Cette URL peut être celle d'un serveur distant
18.
19.    private String getURLWithPort(String uri) {
20.        return URL + port + uri;
  
```



## Paramétrage des tests d'intégration

```
19.     }
20.
21.     // Les tests ici
22. }
```



*Il faut garder à l'esprit que, dans le cadre d'un test d'intégration, tous les services du serveur d'application doivent être démarrés.*

### Quelques explications de l'annotation:

**@SpringBootTest(webEnvironment=WebEnvironment.RANDOM\_PORT, classes={Controller.class, AnotherController.class})**

#### Cette annotation a pour rôle :

- de déclencher le démarrage du serveur d'application sur un port libre ;
- de créer une ApplicationContext pour les tests ;
- d'initialiser une servlet embarquée ;
- et enfin d'enregistrer un bean TestRestTemplate pour la gestion des requêtes HTTP.

**@TestPropertySource(locations = "classpath:test.properties")**. Cette annotation charge le fichier de propriétés contenant toutes les informations nécessaires au déroulement complet des tests (configuration du profil de test, de la base de données, des logs, etc.). On peut tout à fait utiliser les paramètres par défaut de Spring Boot sans avoir besoin de configurer ce fichier.

On peut aussi ajouter l'annotation **@Sql({classpath : init-data.sql})**, C'est le fichier optionnel d'initialisation de la base de données situé dans le classpath. Si ce fichier est placé au niveau de classe, alors, il s'exécute avant chaque cas de test.

**TestRestTemplate** est l'injection d'une dépendance enregistrée par **@SpringBootTest** pour écrire les requêtes HTTP. Vous pouvez configurer les logs et l'accès à la base de données dans le fichier **test.properties** afin d'utiliser une base de données différente de celle de l'application.

Je mets à votre disposition la configuration de la base de données pour les tests à travers le fichier **test.properties**

## test.properties

```
1.
2. # Server
3. #important du contexte sinon erreur
4. server.contextPath = /springboot-restserver
5.
6. security.basic.enabled=false
7.
8. # Database
9. spring.datasource.driverClassName=org.hsqldb.jdbcDriver
10. spring.datasource.url=jdbc:hsqldb:mem:testdb
11. spring.datasource.username=sa
12. spring.datasource.password=
13.
14. # JPA
15. spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.HSQLDialect
16. spring.jpa.hibernate.ddl-auto=create-drop
17. spring.jpa.show-sql=true
18.
19. # Logging
20. logging.level.com.bnguimgo.springbootrestserver=DEBUG
21. logging.level.org.springframework.web.client.RestTemplate=DEBUG
```



Notez le paramétrage du contexte **server.contextPath = /springboot-restserver** de l'application nécessaire utilisé par le contrôleur lors des tests unitaires.

Voici les **tests d'intégration pour tous les services CRUD** à créer dans un package **com.bnguimgo.springbootrestserver.integrationtest** :

#### Tests d'intégration pour tous les services CRUD

```

1.
2. package com.bnguimgo.springbootrestserver.integrationtest;
3.
4. import static org.junit.Assert.assertEquals;
5. import static org.junit.Assert.assertNotNull;
6.
7. import java.util.Collection;
8. import java.util.HashMap;
9. import java.util.Map;
10.
11. import org.junit.Test;
12. import org.junit.runner.RunWith;
13. import org.slf4j.Logger;
14. import org.slf4j.LoggerFactory;
15. import org.springframework.beans.factory.annotation.Autowired;
16. import org.springframework.boot.context.embedded.LocalServerPort;
17. import org.springframework.boot.test.context.SpringBootTest;
18. import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
19. import org.springframework.boot.test.web.client.TestRestTemplate;
20. import org.springframework.http.HttpEntity;
21. import org.springframework.http.HttpMethod;
22. import org.springframework.http.HttpStatus;
23. import org.springframework.http.ResponseEntity;
24. import org.springframework.test.context.TestPropertySource;
25. import org.springframework.test.context.junit4.SpringRunner;
26.
27. import com.bnguimgo.springbootrestserver.model.User;
28.
29. @RunWith(SpringRunner.class)
30. @SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
31. @TestPropertySource(locations = "classpath:test.properties")
32. public class UserControllerIntegrationTest {
33.
34.     private static final Logger logger =
35.         LoggerFactory.getLogger(UserControllerIntegrationTest.class);
36.
37.     @Autowired
38.     private TestRestTemplate restTemplate;
39.
40.     @LocalServerPort //permet d'utiliser le port local du serveur, sinon une erreur "Connection refused"
41.     private int port;
42.     private static final String URL = "http://
43.     localhost:"; //url du serveur REST. Cette URL peut être celle d'un serveur distant
44.
45.     private String getURLWithPort(String uri) {
46.         return URL + port + uri;
47.     }
48.
49.     @Test
50.     public void testFindAllUsers() throws Exception {
51.         ResponseEntity<Object> responseEntity = restTemplate.getForEntity(getURLWithPort("/
52.         springboot-restserver/user/users"), Object.class);
53.
54.         Collection<User> userCollections = (Collection<User>) responseEntity.getBody();
55.         logger.info("Utilisateur trouvé : " + userCollections.toString());
56.
57.         // On vérifie le code de réponse HTTP, en cas de différence entre les deux valeurs, le message "Réponse inattendue"
58.         assertEquals("Réponse inattendue", HttpStatus.FOUND.value(),
59.             responseEntity.getStatusCodeValue());
60.
61.         assertNotNull(userCollections);
62.         assertEquals(3,
63.             userCollections.size()); //on a bien 3 utilisateurs initialisés par les scripts data.sql au démarrage des tests

```

## Tests d'intégration pour tous les services CRUD

```

57.     }
58.
59.     @Test
60.     public void testSaveUser() throws Exception {
61.         User user = new User("PIPO", "password", 1);
62.         ResponseEntity<User> userEntitySaved = restTemplate.postForEntity(getURLWithPort("/springboot-restserver/user/users"), user, User.class);
63.         User userSaved = userEntitySaved.getBody();
64.         assertNotNull(userSaved);
65.         assertEquals(user.getLogin(), userSaved.getLogin());
66.         assertEquals("Réponse inattendue", HttpStatus.CREATED.value(),
        userEntitySaved.getStatusCodeValue());
67.     }
68.     @Test
69.     public void testFindUserByLogin() throws Exception {
70.
71.         Map<String, String> variables = new HashMap<>(1);
72.         variables.put("loginName", "user3");
73.         ResponseEntity<User> responseEntity = restTemplate.getForEntity(getURLWithPort("/springboot-restserver/user/users/{loginName}"), User.class, variables);
74.
75.         User userFound = responseEntity.getBody();
76.         logger.info("Utilisateur trouvé : " + userFound.toString());
77.
78.         // On vérifie le code de réponse HTTP, en cas de différence entre les deux valeurs, le message "Réponse inattendue"
79.         assertEquals("Réponse inattendue", HttpStatus.FOUND.value(),
        responseEntity.getStatusCodeValue());
80.
81.         assertNotNull(userFound);
82.         assertEquals(3, userFound.getId().longValue());
83.     }
84.
85.     @Test
86.     public void testDeleteUser() throws Exception {
87.         Map<String, Long> variables = new HashMap<>(1);
88.         variables.put("id", new Long(1));
89.         ResponseEntity<Void> responseEntity = restTemplate.exchange(getURLWithPort("/springboot-restserver/user/users/{id}"),
90.             HttpMethod.DELETE,
91.             null,
92.             Void.class,
93.             variables);
94.         assertEquals("Réponse inattendue", HttpStatus.GONE.value(),
        responseEntity.getStatusCodeValue());
95.     }
96.
97.     @Test
98.     public void testUpdateUser() throws Exception {
99.         Map<String, Long> variables = new HashMap<>(2);
100.        User
        userToUpdate = new User("updateLogin", "password", 0); // on met à jour l'utilisateur qui a l'identifiant 1
101.        variables.put("id", new Long(2)); // on va désactiver l'utilisateur qui a l'identifiant 2
102.        HttpEntity<User> requestEntity = new HttpEntity<User>(userToUpdate);
103.        ResponseEntity<User> responseEntity = restTemplate.exchange(getURLWithPort("/springboot-restserver/user/users/{id}"),
104.            HttpMethod.PUT,
105.            requestEntity,
106.            User.class, variables);
107.        assertEquals("Réponse inattendue", HttpStatus.OK.value(),
        responseEntity.getStatusCodeValue());
108.    }
109. }

```



*L'annotation **@LocalServerPort** est très importante, car elle permet d'ouvrir automatiquement un port local sur le serveur, par exemple le port **8080**. Sans cette annotation, vous aurez une erreur du type : **Connection refused**.*

Dans la **deuxième partie**, je vais développer un **client web** pour consommer ces services. J'utiliserai pour ce faire **Spring RestTemplate** pour établir la communication entre le client et le serveur. C'est d'ailleurs pour cette que j'ai utilisé Spring RestTemplate pour les tests d'intégration.

Je vous invite à la lire la deuxième partie de ce tutoriel, car la plupart des tutoriels qui existent sur internet ne montrent pas comment mettre en place le client afin de consommer les services. J'ai tenu à bien séparer les deux parties pour démontrer qu'on peut tout à fait développer et consommer ces services sur un serveur distant.

## II - Consommation des services

### II-A - Introduction

Dans cette deuxième partie du tutoriel, je vais développer un **client REST** en utilisant le Framework **Spring RestTemplate** et **SpringMVC**. Le service à consommer par le client a été développé dans la première partie et est **disponible ici**.

**Spring RestTemplate** est un Framework de Spring qui permet d'établir une communication entre un **client** et un **serveur REST**, ceci grâce aux requêtes **HTTP**.

**SpringMVC** permet de faire le lien entre le contrôleur et les pages **JSP** grâce aux mappings des **objets Models (Model, Map, ModelAndView)**.

Le **contrôleur** intercepte toutes les requêtes et les transmet (grâce à Spring RestTemplate) ensuite au service REST qui lui renvoie une réponse, et c'est cette réponse qui est remise au Model à travers une variable pour affichage dans une page JSP.

### II-B - Création du projet client

Le socle projet sera créé par génération depuis le site web de Spring Boot dont voici le lien : <https://start.spring.io/>. Vous pouvez néanmoins créer manuellement un projet Maven classique et le compléter. Avant de créer le socle projet, nous savons que notre application est une application web.

**Voici donc les dépendances nécessaires :**

- **Web** : permet à Spring Boot de rapatrier toutes les dépendances pour une application web (SpringMVC, SpringContext etc.).
- **Packaging : War**, car l'application sera déployée sur un serveur d'application.
- **Java Version 8**.
- Pour accéder à tous les modules fournis par Spring Boot, cliquez sur **Switch to full version**.

**SPRING INITIALIZR** bootstrap your application now

Generate a Maven Project with Java and Spring Boot 1.5.10

### Project Metadata

Artifact coordinates

**Group**

**Artifact**

**Name**

**Description**

**Package Name**

**Packaging**

War

**Java Version**

8

Too many options? [Switch back to the simple version.](#)

### Dependencies

Add Spring Boot Starters and dependencies to your application

**Search for dependencies**

**Selected Dependencies**

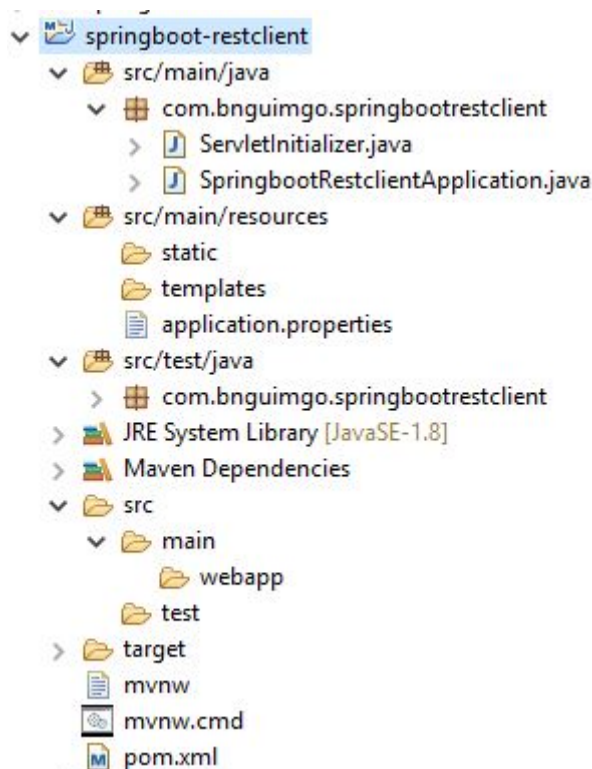
Web

Generate Project alt + ⌘

### Génération Spring Boot Client Rest

Le résultat de la génération est un fichier zip **springboot-restclient.zip** que vous devez décompresser et importer dans votre IDE préféré (dans mon cas, il s'agit d'Eclipse) en suivant la procédure suivante :

**Fichier --> Import ... --> Existing Maven Projects --> Next --> Indiquer le chemin au pom.xml --> Finish**



Structure projet client généré par Spring Boot

Voici le contenu des deux classes générées :

#### ServletInitializer

```
1.
2. package com.bnguimgo.springbootrestclient;
3.
4. import org.springframework.boot.builder.SpringApplicationBuilder;
5. import org.springframework.boot.web.support.SpringBootServletInitializer;
6.
7. public class ServletInitializer extends SpringBootServletInitializer {
8.
9.     @Override
10.    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
11.        return application.sources(SpringbootRestclientApplication.class);
12.    }
13. }
```

La classe **ServletInitializer** étend **SpringBootServletInitializer** qui permet de déployer l'application en tant qu'application web. Elle initialise également la classe de démarrage.

#### SpringbootRestclientApplication

```
1.
2. package com.bnguimgo.springbootrestclient;
3.
4. import org.springframework.boot.SpringApplication;
5. import org.springframework.boot.autoconfigure.SpringBootApplication;
6.
7. @SpringBootApplication
8. public class SpringbootRestclientApplication {
9.
10.    public static void main(String[] args) {
11.        SpringApplication.run(SpringbootRestclientApplication.class, args);
12.    }
13. }
```

La classe **SpringbootRestclientApplication** est le point d'entrée de l'application, et possède à cet effet la méthode **main(String[] args)** qui exécute l'application au démarrage

Il faut compléter le **pom.xml** avec les propriétés et dépendances manquantes. Dans la partie propriété, j'ai indiqué quel est le point d'entrée de l'application, et la version de Tomcat. Ces deux configurations restent optionnelles, car Spring boot sait se débrouiller tout seul. Mais, comme je vais utiliser un Tomcat externe, il faut renseigner la version de Tomcat. Vous pouvez donc supprimer la dépendance de Tomcat ajoutée par défaut lors de la génération.

```
1.
2. <properties>
3. <start-class>com.bnguimgo.springbootrestclient.SpringbootRestclientApplication</start-class><!--
   cette déclaration est optionnelle -->
4. <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
5. <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
6. <java.version>1.8</java.version>
7. <tomcat.version>8.0.41</tomcat.version><!-- Permet de modifier la version de Tomcat
   embarquée -->
8. </properties>
```

Comme on va développer les pages web en utilisant la technologie **JSP**, il faut ajouter la dépendance **jstl**, car Spring Boot ne sait pas d'avance quel client web il va supporter. Cependant, Spring Boot intègre un Framework de Template Thymeleaf que je ne vais pas utiliser dans le cadre de ce projet.

On va donc ajouter cette dépendance :

```
1.
2. <dependency>
3. <groupId>javax.servlet</groupId>
4. <artifactId>jstl</artifactId>
5. </dependency>
```



*Pensez à faire de temps en temps une synchronisation des dépendances dans Eclipse en faisant un **clic droit --> Maven --> Update Projects ...***

J'ai supprimé les dépendances **spring-boot-starter-tomcat** et **spring-boot-starter-test** puisque nous n'en aurons pas besoin. Il faut aussi supprimer le package de tests. Pour les tests unitaires et tests d'intégration, se référer à la **première partie** de ce tutoriel.

Voici le **pom.xml final** de l'application :

#### pom.xml final

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.bnguimgo</groupId>
  <artifactId>springboot-restclient</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>war</packaging>

  <name>springboot-restclient</name>
  <description>Client REST utilisant le framework Spring Boot</description>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.5.10.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
```

#### pom.xml final

```
<properties>
  <start-class>com.bnquimgo.springbootrestclient.SpringbootRestclientApplication</start-
class><!-- cette déclaration est optionnelle -->
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
  <java.version>1.8</java.version>

  <tomcat.version>8.0.41</tomcat.version><!-- Permet de modifier la version de Tomcat embarquée --
>
</properties>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <!-- obligatoire: JSTL pour les pages JSP, car Spring Boot ne sais pas quel type de client vous allez utiliser
>
  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
  </dependency>
</dependencies>

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

En résumé, le pom.xml ne contient que deux dépendances :

- Une dépendance **spring-boot-starter-web** pour la partie web.
- Une dépendance **JSTL** pour les servlets et pages JSP.

Il faut exécuter un premier build pour vérifier que la configuration est bonne.

## II-C - Création du contrôleur

Je vais créer un contrôleur par défaut dont le rôle est d'envoyer une requête ping au serveur pour vérifier si le serveur est démarré et donc disponible, sinon, une page d'erreur est affichée.

### II-C-1 - Contrôleur par défaut

Créer un package **controller** et y ajouter la classe **WelcomeController**.

#### contrôleur WelcomeController

```
1.
2. package com.bnquimgo.springbootrestclient.controller;
3. import org.slf4j.Logger;
4. import org.slf4j.LoggerFactory;
5. import org.springframework.beans.factory.annotation.Autowired;
6. import org.springframework.http.ResponseEntity;
7. import org.springframework.stereotype.Controller;
8. import org.springframework.web.bind.annotation.GetMapping;
9. import org.springframework.web.client.RestTemplate;
10. import org.springframework.web.servlet.ModelAndView;
```



### contrôleur WelcomeController

```

11.
12. @Controller
13. public class WelcomeController {
14.
15.     private static final Logger logger = LoggerFactory.getLogger(WelcomeController.class);
16.
17.     @Autowired
18.     private RestTemplate restTemplate;
19.
20.     @Autowired
21.     private PropertiesConfigurationService configurationService ;
22.
23.     @GetMapping(value="/")
24.     ModelAndView ping(ModelAndView modelAndView) {
25.
26.         ResponseEntity<String> reponseServeur =
27.             restTemplate.getForEntity(configurationService.getUrl(), String.class);
28.         int codeReponseServeur= reponseServeur.getStatusCodeValue();
29.         String reponsePing="";
30.         if (codeReponseServeur!=200) {
31.             logger.error("Réponse du serveur: "+codeReponseServeur+" ==> Serveur indisponible, votre application ne fonctionne pas");
32.             reponsePing=configurationService.getPingServeurKo();
33.         }else{
34.             reponsePing=configurationService.getPingServeurOk();
35.             logger.info(configurationService.getPingServeur(),reponsePing);
36.         }
37.         //construction de la vue
38.         modelAndView.setViewName("welcome");
39.         modelAndView.addObject("urlServeur", configurationService.getUrl());
40.         modelAndView.addObject("pingServeur", reponsePing);
41.         modelAndView.addObject("profileActif", configurationService.getProfileActif());
42.         return modelAndView;
43.     }
44.     @GetMapping(value = "/error")
45.     public String error() {
46.         return "error";
47.     }
48.
49.     @GetMapping("/next")
50.     ModelAndView next(ModelAndView modelAndView) {
51.         modelAndView.setViewName("next");
52.         modelAndView.addObject("message", configurationService.getMessage());
53.         return modelAndView;
54.     }
55. }

```

L'injection par `@Autowired` de **RestTemplate** va permettre de construire les requêtes **HTTP** nécessaires pour établir la communication avec le serveur.

Grâce à l'annotation `@GetMapping(value="/")`, la méthode **ModelAndView ping(ModelAndView modelAndView)** {xxx } envoie une requête "ping" au serveur à l'adresse **app.serveur.url = http://localhost:8080/springboot-restserver/**. Cette requête est construite avec **RestTemplate**

Si la réponse reçue correspond au code **HttpStatus = 200**, c'est que le serveur est disponible. On construit ensuite la vue **welcome.jsp** grâce à la classe **ModelAndView** fournie par Spring en lui passant en paramètres la page à afficher, l'**URL du serveur**, la **réponse du serveur**, et enfin le **profil utilisateur**.

Si le serveur ne répond pas, une page d'erreur **error.jsp** sera construite et renvoyée grâce à la méthode :

### code de la page d'erreur

```

1.
2. @GetMapping(value = "/error")
3. public String error() {
4.     return "error";

```

## code de la page d'erreur

```
5.    }
```

L'adresse du serveur est configurée dans le fichier **application.properties** et accessible grâce à la classe utilitaire ci-dessous.

## Classe utilitaire PropertiesConfigurationService

```
1.
2. package com.bnnguingo.springbootrestclient.controller;
3.
4. import org.springframework.beans.factory.annotation.Value;
5. import org.springframework.stereotype.Service;
6.
7. @Service
8. public class PropertiesConfigurationService{
9.
10.     @Value("${app.serveur.url}") // injection via application.properties
11.     private String url="";
12.
13.     @Value("${app.serveur.pingServeur}")
14.     private String pingServeur;
15.
16.     @Value("${app.serveur.ok}")
17.     private String pingServeurOk;
18.
19.     @Value("${app.serveur.ko}")
20.     private String pingServeurKo;
21.
22.     @Value("${nextpage.message}")
23.     private String message;
24.
25.     @Value("${spring.profiles.active}")
26.     private String profileActif;
27.     // getters, setters
28.
29.     public String getUrl() {
30.         return url;
31.     }
32.
33.     public String getPingServeur() {
34.         return pingServeur;
35.     }
36.
37.     public String getPingServeurOk() {
38.         return pingServeurOk;
39.     }
40.
41.     public String getPingServeurKo() {
42.         return pingServeurKo;
43.     }
44.
45.     public String getMessage() {
46.         return message;
47.     }
48.
49.     public String getProfileActif() {
50.         return profileActif;
51.     }
52. }
```

Cette classe utilitaire **PropertiesConfigurationService** permet de récupérer l'URL du serveur stockée dans le fichier de configuration **application.properties**. Cette classe permet également de récupérer toutes les autres variables du fichier de configuration. Le principal avantage de cette classe, c'est qu'elle permet d'éviter qu'une même variable soit injectée dans plusieurs classes, ce qui rend la maintenance facile.

Voici les trois fichiers **.properties** à ajouter dans **source/main/resources/**

### Le fichier application.properties contient les propriétés globales de l'application

```

1.
2. #Au démarrage de l'application, en cas d'erreur
3. server.error.whitelabel.enabled = false
4. #Chargement du profil par défaut dev
5. spring.profiles.active=dev
  
```

### Le fichier application-dev-properties contient la configuration nécessaire en phase de développement

```

1.
2.
3. #URL du serveur REST
4. app.serveur.url = http://localhost:8080/springboot-restserver/
5. nextpage.message = Salut vous &ecirc;tes en profile dev
6. app.serveur.ok = disponible
7. app.serveur.ko = indisponible
8. app.serveur.pingServeur = Le serveur est {}
9. compte.user.exist = Un utilisateur est d&eacute;j&agrave; enregistr&eacute; avec ce compte
10.
11. #GESTION DES LOGS
12. logging.level.org.springframework.web=DEBUG
13. logging.level.com.bnguimgo.restclient=DEBUG
14.
15. # Pattern impression des logs consoles
16. logging.pattern.console= %d{yyyy-MM-dd HH:mm:ss} - %msg%n
17.
18. # Pattern impression des logs dans un fichier
19. logging.pattern.file= %d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg%n
20. # Redirection des logs vers un fichier du repertoire Temp, exemple sur windows: C:\Users
   \UserName\AppData\Local\Temp\
21. logging.file= ${java.io.tmpdir}\logs\restClient\applicationRestClient.log
  
```

### Le fichier application-prod-properties contient la configuration nécessaire en phase de production

```

1.
2.
3. logging.level.org.springframework.web=ERROR
4. logging.level.com.bnguimgo.restclient=ERROR
5. #
6. # Pattern impression des logs console
7. logging.pattern.console= %d{yyyy-MM-dd HH:mm:ss} - %msg%n
8.
9. # Pattern impression des logs dans un fichier
10. logging.pattern.file= %d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg%n
11. # Redirection des logs vers un fichier du repertoire Temp, exemple sur windows: C:\Users
   \UserName\AppData\Local\Temp\
12. logging.file= ${java.io.tmpdir}/logs/rest/applicationRest.log
  
```

## II-C-2 - Contrôleur UserController

Ce contrôleur gère tous les services CRUD relatifs au compte utilisateur. Je vais mettre en œuvre deux de ces services : la **création** et la **connexion** d'un utilisateur.

### UserController

```

1.
2.
3. package com.bnguimgo.springbootrestclient.controller;
4. import java.util.HashMap;
5. import java.util.Map;
6.
7. import javax.validation.Valid;
8.
9. import org.slf4j.Logger;
10. import org.slf4j.LoggerFactory;
11. import org.springframework.beans.factory.annotation.Autowired;
12. import org.springframework.http.ResponseEntity;
13. import org.springframework.stereotype.Controller;
14. import org.springframework.validation.BindingResult;
15. import org.springframework.web.bind.annotation.GetMapping;
  
```

## UserController

```
16. import org.springframework.web.bind.annotation.ModelAttribute;
17. import org.springframework.web.bind.annotation.PostMapping;
18. import org.springframework.web.client.RestTemplate;
19. import org.springframework.web.servlet.ModelAndView;
20.
21. import com.bnquingo.springbootrestclient.dto.UserDto;
22. import com.bnquingo.springbootrestclient.dto.UserRegistrationForm;
23. import com.bnquingo.springbootrestclient.entities.User;
24.
25.
26. @Controller
27. public class LoginController {
28.
29.     private static final Logger logger = LoggerFactory.getLogger(LoginController.class);
30.
31.     @Autowired
32.     private RestTemplate restTemplate;
33.
34.     @Autowired
35.     private PropertiesConfigurationService configurationService ;
36.
37.     @GetMapping(value = "/login")//initialisation du login
38.     public String loginView(Map<String, Object> model) {
39.         UserDto userDto = new UserDto();
40.         model.put("userForm", userDto);
41.         return "loginForm";
42.     }
43.
44.     @PostMapping(value = "/login")
45.     public ModelAndView login(@Valid @ModelAttribute("userForm") UserDto userForm,
46.         BindingResult bindingResult, ModelAndView modelAndView) {
47.
48.         modelAndView.setViewName("loginForm");
49.         if (bindingResult.hasErrors()) {
50.             return modelAndView;
51.         }
52.         Map<String, String> variables = new HashMap<String, String>(1);
53.         variables.put("loginName", userForm.getLogin());
54.
55.         ResponseEntity<User> userExists =
56.             restTemplate.getForEntity(configurationService.getUrl() + "user/users/{loginName}", User.class,
57.                 variables);
58.         User user = userExists.getBody();
59.         if (user == null) { //ceci nous évite d'écrire une page de gestion des erreurs
60.             modelAndView.addObject("saveError", "Aucun utilisateur trouvé avec ce compte");
61.             return modelAndView;
62.         }
63.         modelAndView.setViewName("loginSuccess");
64.         return modelAndView;
65.
66.     @GetMapping(value = "/registration")//initialisation du formulaire de création du compte
67.     public String registrationView(Map<String, Object> model) {
68.         UserRegistrationForm userRegistrationForm = new UserRegistrationForm();
69.         model.put("registrationForm", userRegistrationForm);
70.         return "registrationForm";
71.     }
72.
73.     @PostMapping(value = "/registration")
74.     public ModelAndView saveUser(@Valid @ModelAttribute("registrationForm")
75.         UserRegistrationForm userRegistrationForm,
76.         BindingResult bindingResult, ModelAndView modelAndView) {
77.         modelAndView.setViewName("registrationForm");
78.         if (bindingResult.hasErrors()) {
79.             return modelAndView;
80.         }
81.         Map<String, String> variables = new HashMap<String, String>(1);
82.         variables.put("loginName", userRegistrationForm.getLogin());
```

## UserController

```

82.         ResponseEntity<User> userEntity =
            restTemplate.getForEntity(configurationService.getUrl() + "user/users/{loginName}", User.class,
                variables);
83.         User userExists = userEntity.getBody();
84.         if (userExists != null) { //ceci nous évite d'écrire une page de gestion des erreurs
85.             logger.info("userExists", userExists.toString());
86.
            bindingResult.rejectValue("login", "error.user", "Un utilisateur est déjà enregistré avec ce compte");//Note: la
87.             return modelAndView;
88.         }
89.
90.         //return "loginSuccess";
91.         ResponseEntity<User> userEntitySaved =
            restTemplate.postForEntity(configurationService.getUrl() + "user/
users", new User(userRegistrationForm),
            User.class); //point de liaison entre le client REST et le serveur REST grace à RestTemplate
92.         User userSaved = userEntitySaved.getBody();
93.         if (null == userSaved) {
94.             modelAndView.addObject("saveError", "erreur de création du compte.");
95.             return modelAndView;
96.         }
97.         modelAndView.setViewName("loginSuccess");
98.         modelAndView.addObject("userSaved", userSaved);
99.         return modelAndView;
100.     }
101. }
  
```

La méthode annotée par **@PostMapping(value = "/login")** permet de traiter les requêtes de connexion à l'application.

La méthode annotée par **@PostMapping(value = "/registration")** permet de créer le compte utilisateur. Cette méthode utilise une instance de l'objet `restTemplate` pour construire la requête vers le serveur.

## II-C-3 - Configuration de l'application

Dans cette section, je vais créer une classe de configuration **MvcConfiguration** juste pour indiquer à Spring Boot où se trouvent les pages web à charger, puisque je ne vais pas utiliser la configuration par défaut qui consiste à mettre toutes les pages statiques dans le dossier **source/main/resources/static généré**. Créer un package **config** et y ajouter cette classe.

## MvcConfiguration

```

1.
2. package com.bnguimgo.springbootrestclient.config;
3.
4. import org.springframework.boot.web.client.RestTemplateBuilder;
5. import org.springframework.context.annotation.Bean;
6. import org.springframework.context.annotation.ComponentScan;
7. import org.springframework.context.annotation.Configuration;
8. import org.springframework.web.client.RestTemplate;
9. import org.springframework.web.servlet.config.annotation.DefaultServletHandlerConfigurer;
10. import org.springframework.web.servlet.config.annotation.EnableWebMvc;
11. import org.springframework.web.servlet.config.annotation.ViewResolverRegistry;
12. import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;
13. import org.springframework.web.servlet.view.InternalResourceViewResolver;
14. import org.springframework.web.servlet.view.JstlView;
15.
16. @Configuration
17. @EnableWebMvc
18. @ComponentScan
19. public class MvcConfiguration extends WebMvcConfigurerAdapter {
20.
21.     @Override
22.     public void configureViewResolvers(ViewResolverRegistry registry) {
23.         InternalResourceViewResolver resolver = new InternalResourceViewResolver();
24.         resolver.setPrefix("/WEB-INF/views/");
  
```

## MvcConfiguration

```

25.     resolver.setSuffix(".jsp");
26.     resolver.setOrder(1);
27.     resolver.setViewClass(JstlView.class);
28.     registry.viewResolver(resolver);
29. }
30. @Override
31. public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {
32.     configurer.enable();
33. }
34.
35. @Bean
36. public RestTemplate restTemplate(RestTemplateBuilder builder) {
37.     builder.setConnectTimeout(10); //set timeout connection
38.     return builder.build();
39. }
40. }

```

D'après cette configuration, toutes les pages web (**vues**) seront dans le dossier **"/WEB-INF/views/"**.

L'annotation **@Configuration** indique que cette classe peut être utilisée par le conteneur Spring comme une source de définition des beans. On constate par ailleurs que le bean **restTemplate** y a été initialisé.



*Si on ne déclarait pas le bean **restTemplate**, il serait impossible d'injecter RestTemplate par @Autowired dans le contrôleur WelcomeController, à défaut il faut faire un **new RestTemplate()** à chaque appel de RestTemplate.*



*Depuis **Spring-4**, on ne peut plus injecter directement RestTemplate par @Autowired car, la classe RestTemplate a été migrée dans **RestTemplateBuilder** qui offre plus de méthodes, comme la configuration du timeout, etc.*

L'annotation **@EnableWebMvc** est l'équivalent de **mvc:annotation-driven** si on utilisait la configuration **XML**. Cette annotation active le contexte SpringMVC, et permet d'utiliser les annotations **@Controller**.

**@ComponentScan** scanne par défaut toutes les classes du même package pour trouver les classes annotées.

## II-C-4 - Création des pages web

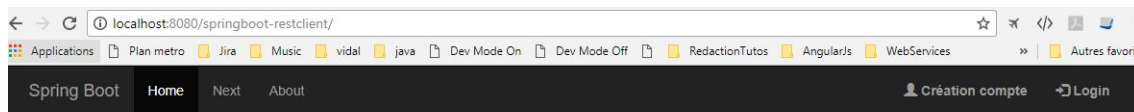
Créer les dossiers **WEB-INF/views/** dans **webapp** pour y ajouter les **vues.jsp**. Toutes les vues que j'ai créées sont disponibles en téléchargement dans le répertoire **WEB-INF/views/**.

Les vues (pages web) et toutes les sources sont  **disponibles en téléchargement**.

## II-D - Test de l'application

Commencer par démarrer le serveur, ensuite le client (clic droit sur le projet --> Run As --> Run On Server --> Choisir Tomcat-8). Voici l'URL de test : <http://localhost:8080/springboot-restclient/>.

## II-D-1 - Serveur démarré



### Exemple de Client REST avec Spring Boot

URL du serveur : <http://localhost:8080/springboot-restserver/>

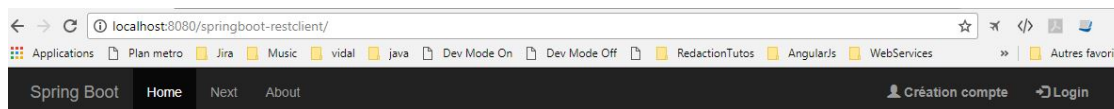
Etat du serveur : disponible

Profile utilisateur : dev

*Serveur REST démarré et à l'écoute*

## II-D-2 - Serveur arrêté

Si on démarre le client alors que le **serveur est arrêté**, on aura l'erreur ci-dessous:

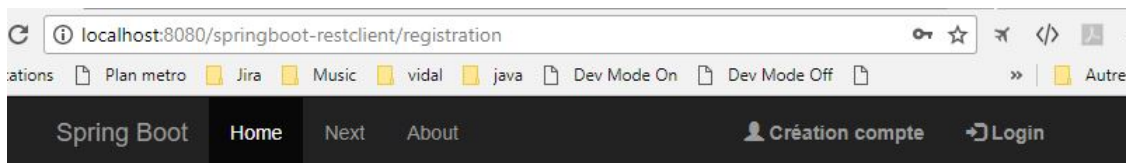


Une erreur est survenue !!!!!.

[retour à l'accueil](#)

*Serveur REST arrêté*

## II-D-3 - Création d'un utilisateur



### Créez votre compte

Enregistrez-vous

Login:

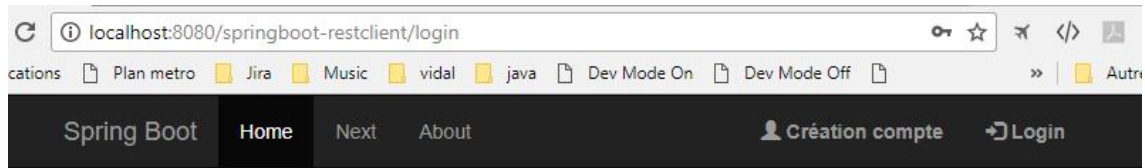
Password:

Enregistrer

*Écran création compte utilisateur*



## II-D-4 - Test de connexion



### Spring Boot Rest Client - Login Form

Connectez-vous

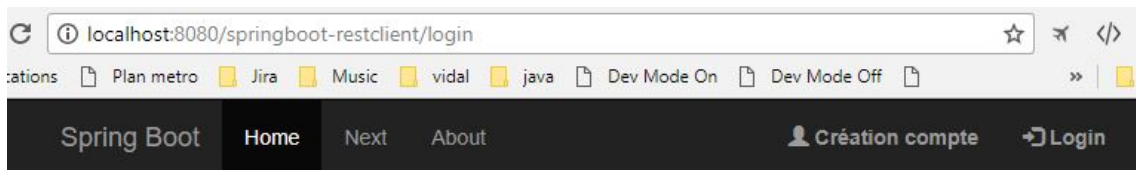
Email:

Password:

Connexion

*Écran de connexion*

## II-D-5 - Page de connexion réussie



Welcome ! You have logged in successfully.

[Deconnexion](#)

*Connexion avec succès*



### III - Conclusion

Ce chapitre sur les services REST s'achève. Nous venons de découvrir l'efficacité de Spring Boot qui nous facilite grandement la tâche en matière de configuration des dépendances, facilite l'instanciation des beans, rend le déploiement d'une application transparent pour le développeur, etc. Spring Boot intègre parfaitement les frameworks de tests unitaires et de tests d'intégration. Grâce à Spring RestTemplate, il est très facile de consommer un service REST.

### III-A - Sources et références

#### Projet Spring Boot

-  <http://projects.spring.io/spring-boot/>

#### Pour générer un projet Spring Boot

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

#### D'excellents tutoriels sur REST et SOAP

-  [http://mbaron.developpez.com/#page\\_soa](http://mbaron.developpez.com/#page_soa)




#### Mise en place des filtres pour la gestion des requêtes Cross Domain

-  <http://sqli.developpez.com/tutoriels/spring/construire-backend-springboot/>

#### Étude comparée SOAP et REST

-  <https://www.supinfo.com/articles/single/2441-comprendre-soap-rest-leurs-differences>

### III-B - Remerciements

Tous mes remerciements au responsable de la rubrique JAVA ( **M Mickael Baron**) qui m'a suivi et encouragé dès le départ à la rédaction de cet article. La correction orthographique a été assurée par  **M Cedric Duprez**, à qui j'adresse mes remerciements sans fautes. Je remercie également tous les développeurs qui ont travaillé à la mise en place des outils qui facilitent la rédaction et la publication des tutoriels sur  [www.developpez.com](http://www.developpez.com)