

Spring Boot et Rest

**TP N°1: Développement d'une application java (JAR) avec
Spring Boot et Rest**

SOMMAIRE

1.Pré-requis :	3
2.Objectifs :	3
3. Création du squelette de votre projet Maven	3
4.Développement de la classe de démarrage de Spring Boot.....	8
5.Développement du contrôleur HelloController.....	8
6.Build de l'application.....	10
7.Développement du contrôleur ProductController	13
8.Rebuild de l'application	17
9.Tester les services Rest de ProductController	17
10.Produire le format XML.....	20
11.Les fichiers de configuration de l'application de Spring Boot : application-X.properties ..	21
12.Lire une valeur à partir du fichier application-X.properties	23
13.Utiliser un fichier de configuration externe.....	23

1. Pré-requis :

- Eclipse JEE-2020 (ou supérieur) ou tout IDE équivalent (intellij) ;
- JDK 1.8 ou jdk 17;
- Connection à Internet pour permettre à Maven de télécharger les dépendances nécessaires (Spring Boot 2.2.0, ...).
- POSTMAN ou un autre outil pour tester les méthodes POST, PUT et DELETE.

2. Objectifs :

- ✓ Utiliser **Spring Initializer** pour créer le squelette du projet Maven.
- ✓ Ajouter la dépendance web (spring-boot-starter-web) via l'interface de **Spring Initializer**.
- ✓ Utiliser l'annotation **@SpringBootApplication** nécessaire pour la configuration de Spring Boot.
- ✓ Utiliser l'annotation **@RestController** et développer les services CRUD.
- ✓ Packager votre application (rest.jar).
- ✓ Lancer et tester l'application.
- ✓ Comprendre la notion de SPRING ACTIVE PROFILE.
- ✓ Utiliser un fichier externe à l'application.

3. Création du squelette de votre projet Maven

- Aller au site : <https://start.spring.io/> :

The screenshot shows the Spring Initializr interface. The 'Project' section has 'Maven' selected. The 'Language' section has 'Java' selected. The 'Dependencies' section has 'Spring Web' selected. The 'Project Metadata' section shows the following configuration:

Group	com.cigma
Artifact	rest
Name	rest
Description	Project for Spring Boot : HelloWord +Rest + CRUD Produit
Package name	com.cigma.rest

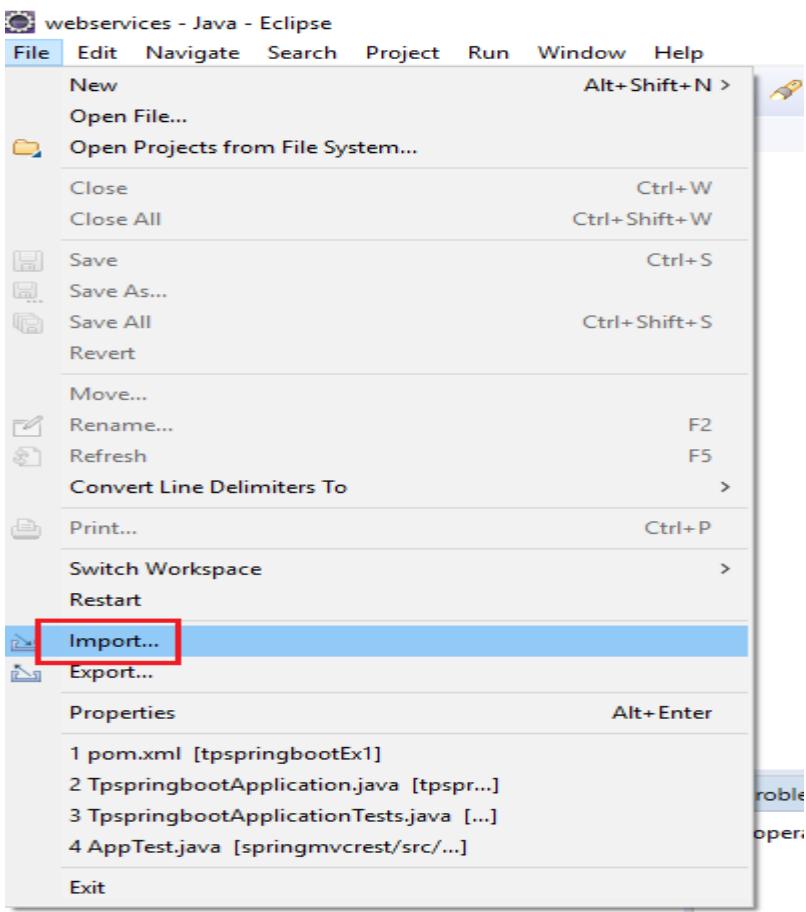
At the bottom, there are three buttons: 'GENERATE' (CTRL + D), 'EXPLORE' (CTRL + SPACE), and 'SHARE...'.

- 1 : Choisir « Maven Project ».
- 2 : Choisir Java.
- 3 : Choisir la version 2.7.3 de Spring Boot (ou bien 3.2.10)

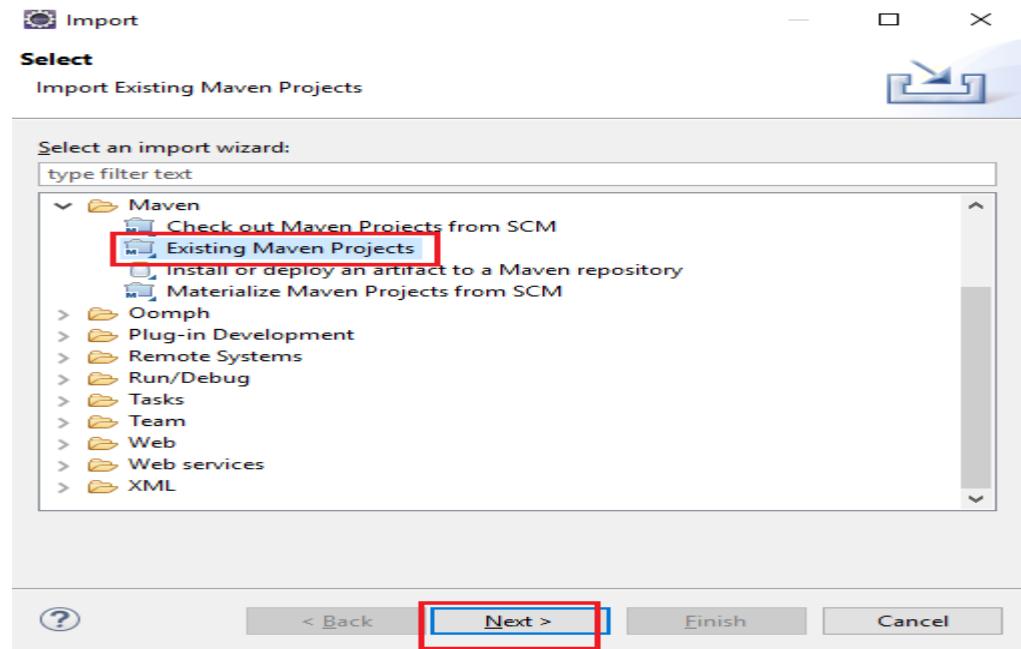
- 4 : Entrer le group (ma.cigma) et l'artifact (rest)
- 5 : Saisir une description de votre application.
- 6 : Saisir le nom du package racine (ici : ma.cigma.rest).
- 7 : Choisir Jar comme packaging.
- 8 : Choisir la version 8 ou 17 de java.
- 9 : Ajouter la dépendance « web ».
- 10 : Enfin cliquer sur le bouton « Generate Projet ». Le fichier ZIP suivant sera généré automatiquement :



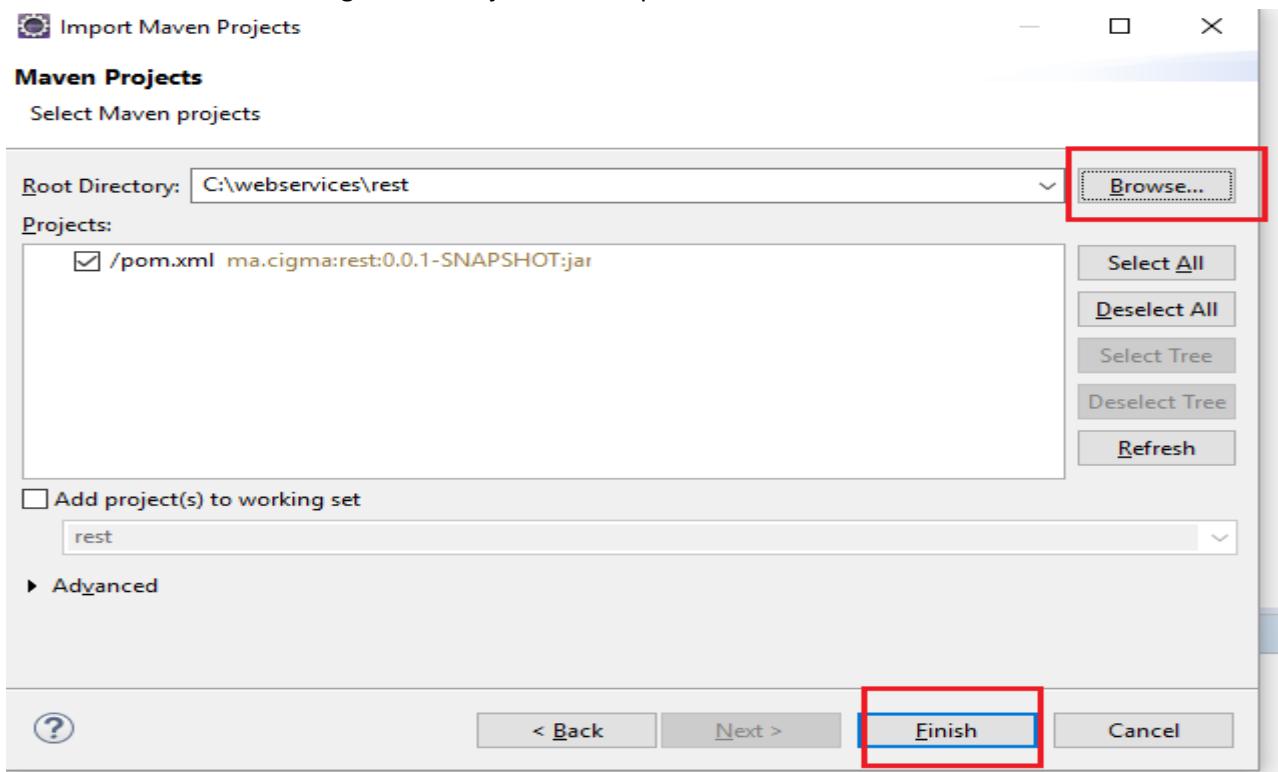
- Accéder au projet (dossier téléchargement)
- Décompresser le fichier rest.zip dans le dossier c:\webservices par exemple, ensuite importer le projet Maven au niveau d'éclipse comme illustré ci-après :



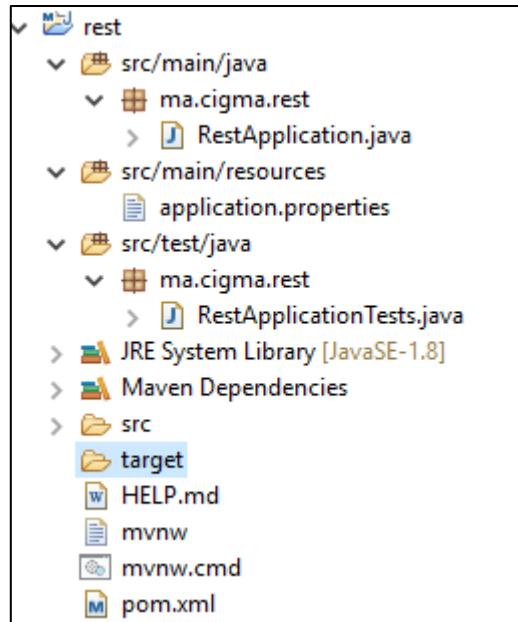
- Cliquer sur le menu « Import... » :



- Choisir « Existing Maven Projects » et cliquer sur Next> :



- Cliquer sur « Browse... » et choisir le répertoire dans lequel existe votre projet Maven (ici : c:\webservices\rest) et cliquer sur Finish. L'arborescence du projet « rest » est comme suit :



* Vérifier les deux dépendances au niveau de votre fichier pom.xml:

```

<?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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.7.17</version>
        <relativePath/> 
    </parent>
    <groupId>com.cigma</groupId>
    <artifactId>rest</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>rest</name>
    <description>Project for Spring Boot : HelloWord +Rest + CRUD Produit</description>
    <properties>
        <java.version>1.8</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

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

```

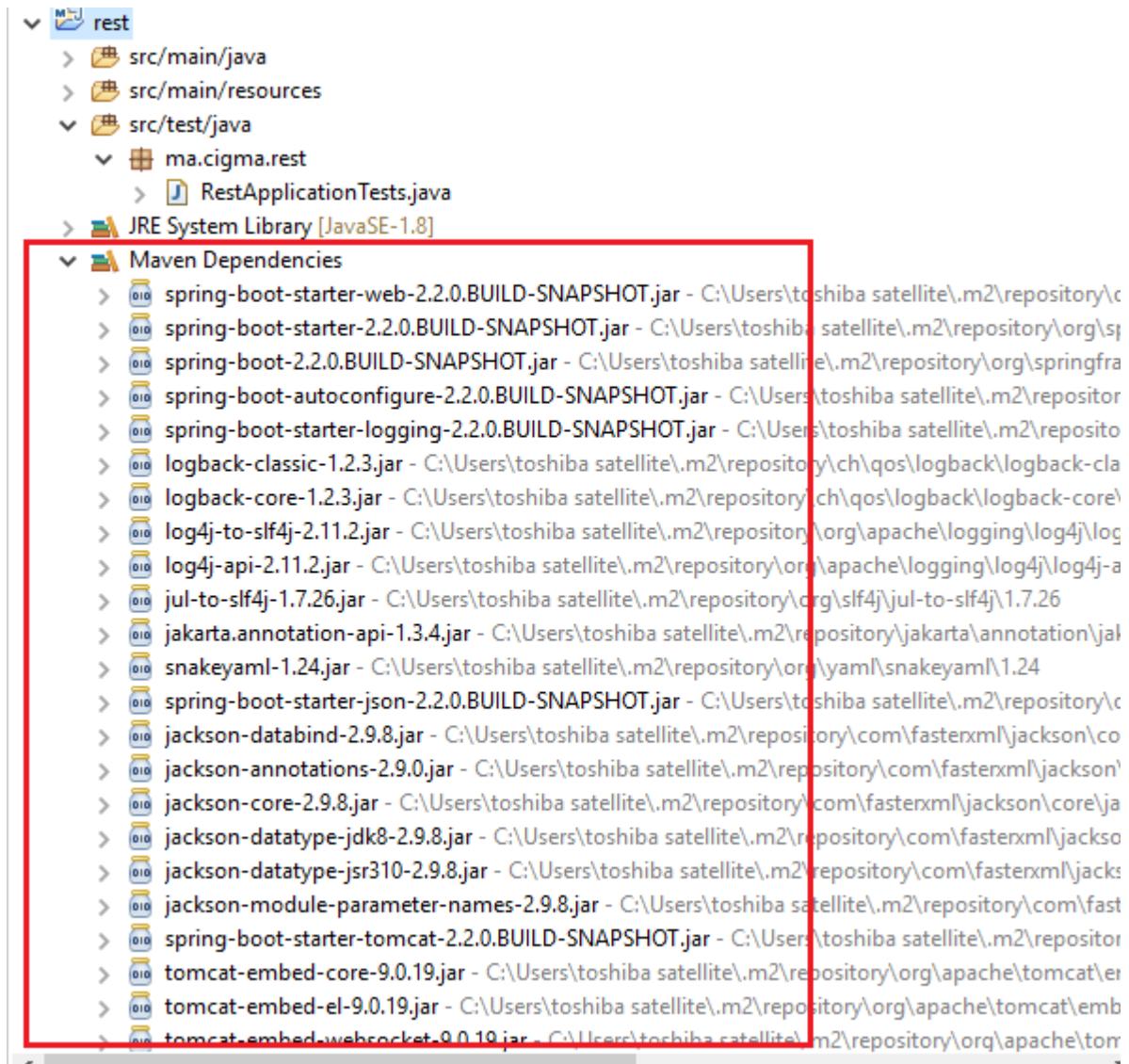
```

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

```

- **spring-boot-starter-web** : est le Starter fourni par Spring Boot qui permet de créer et configurer facilement un projet Rest en gérant toutes les dépendances nécessaires.

*Vérifier que Maven a téléchargé tous les JARS :



4. Développement de la classe de démarrage de Spring Boot

* Remarquez que Spring Boot crée automatiquement la classe RestApplication. Modifier cette comme suit :

```
package ma.cigma.rest;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

/**
 * @SpringBootApplication scanne le classpath, détecte les
 * dépendance puis effectue automatiquement
 * toutes les configurations nécessaires.
 */
@SpringBootApplication
public class RestApplication {
    public static void main(String[] args) {
        SpringApplication.run(MainApplication.class, args);
    }
}
```

Explication :

- Le point d'entrée de l'application Spring Boot est la classe qui contient l'annotation @SpringBootApplication.

@SpringBootApplication

Cette annotation est utilisée dans la classe d'application lors de la configuration d'un projet Spring Boot. La classe annotée avec @SpringBootApplication doit être conservée dans le package de base. La seule chose que fait @SpringBootApplication est une analyse de composant. Mais elle va scanner seulement ses sous-paquets. Par exemple, si vous mettez la classe annotée avec @SpringBootApplication dans com.exemple, alors @SpringBootApplication balayera tous ses sous-packages, tels que com.exemple.a, com.exemple.b et com.exemple.a.x.

Le @SpringBootApplication est une annotation pratique qui ajoute tout ce qui suit:

- @Configuration : Cette annotation est utilisée sur les classes qui définissent les beans. C'est un analogue pour un fichier de configuration XML
- @EnableAutoConfiguration : Cette annotation est généralement placée sur la classe d'application principale. L'annotation @EnableAutoConfiguration définit implicitement un "package de recherche" de base. Cette annotation indique à Spring Boot de commencer à ajouter des beans en fonction des paramètres de chemin de classe, d'autres beans et de divers paramètres de propriété.
- @ComponentScan : Cette annotation est utilisée avec l'annotation @Configuration pour permettre à Spring de connaître les paquets à analyser pour les composants annotés.

5. Développement du contrôleur HelloController

- Créer un nouveau package : ma.cigma.rest.controller
- Développer la class HelloController

```
package ma.cigma.rest.controller;
//Rappel : ce package sera scané par @SpringBootApplication

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloController {

    @RequestMapping(value = { "/hello", "/" })
    public String hello() {

        System.out.println("ce package sera scané par
@SpringBootApplication");

        return "Hello World From my first API
@RestController";
    }
}
```

Par contre la classe ci-dessous ne sera pas scannée par @SpringBootApplication

```
package com.cigma.rest.controller;
//Attention : ce package ne sera pas scané par @SpringBootApplication
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloController2 {

    @RequestMapping(value = { "/hello2", "/h2" })
    public String hello() {

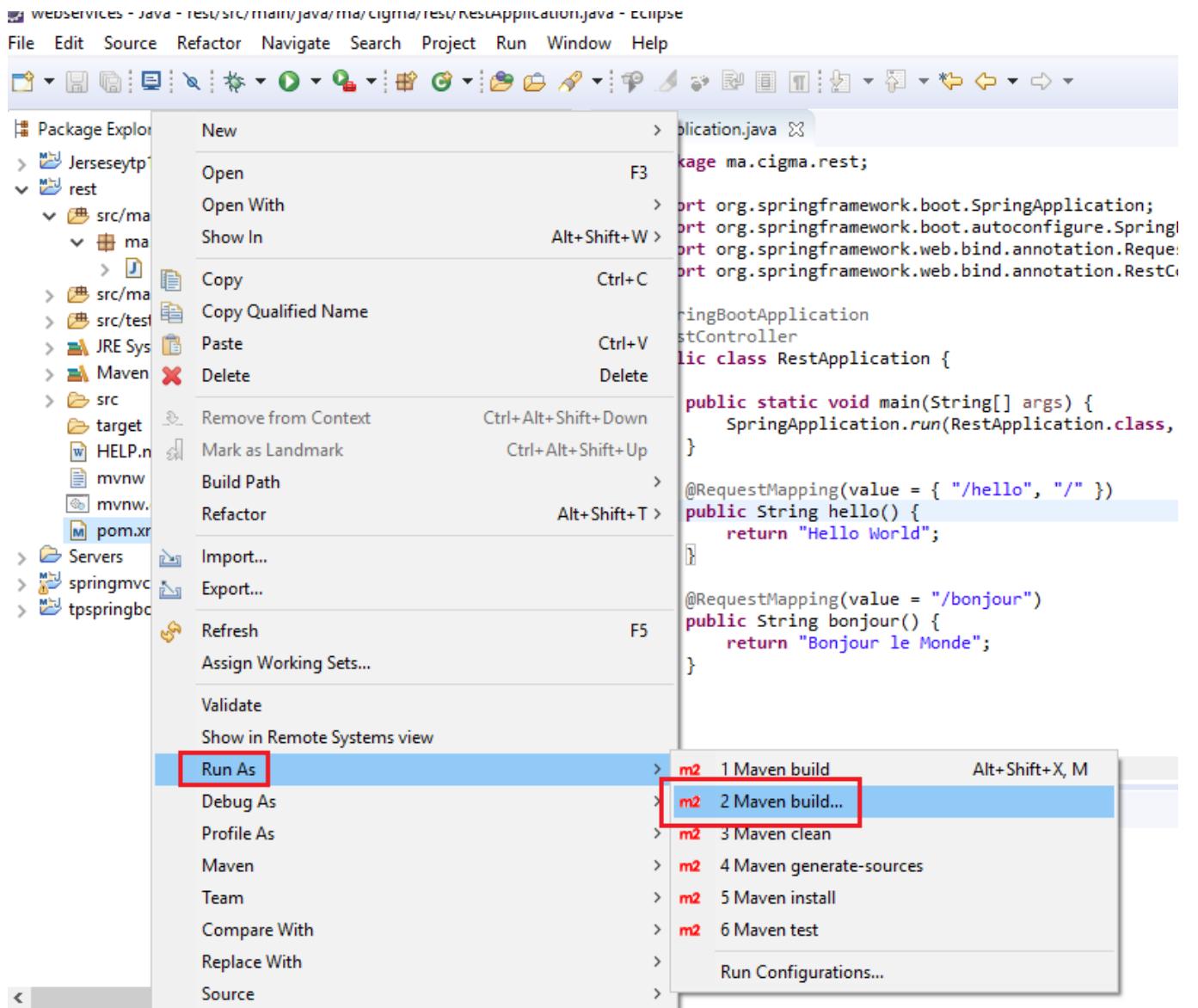
        System.out.println("ce package sera scané par
@SpringBootApplication");

        return "Hello World From my first API
@RestController";
    }
}
```

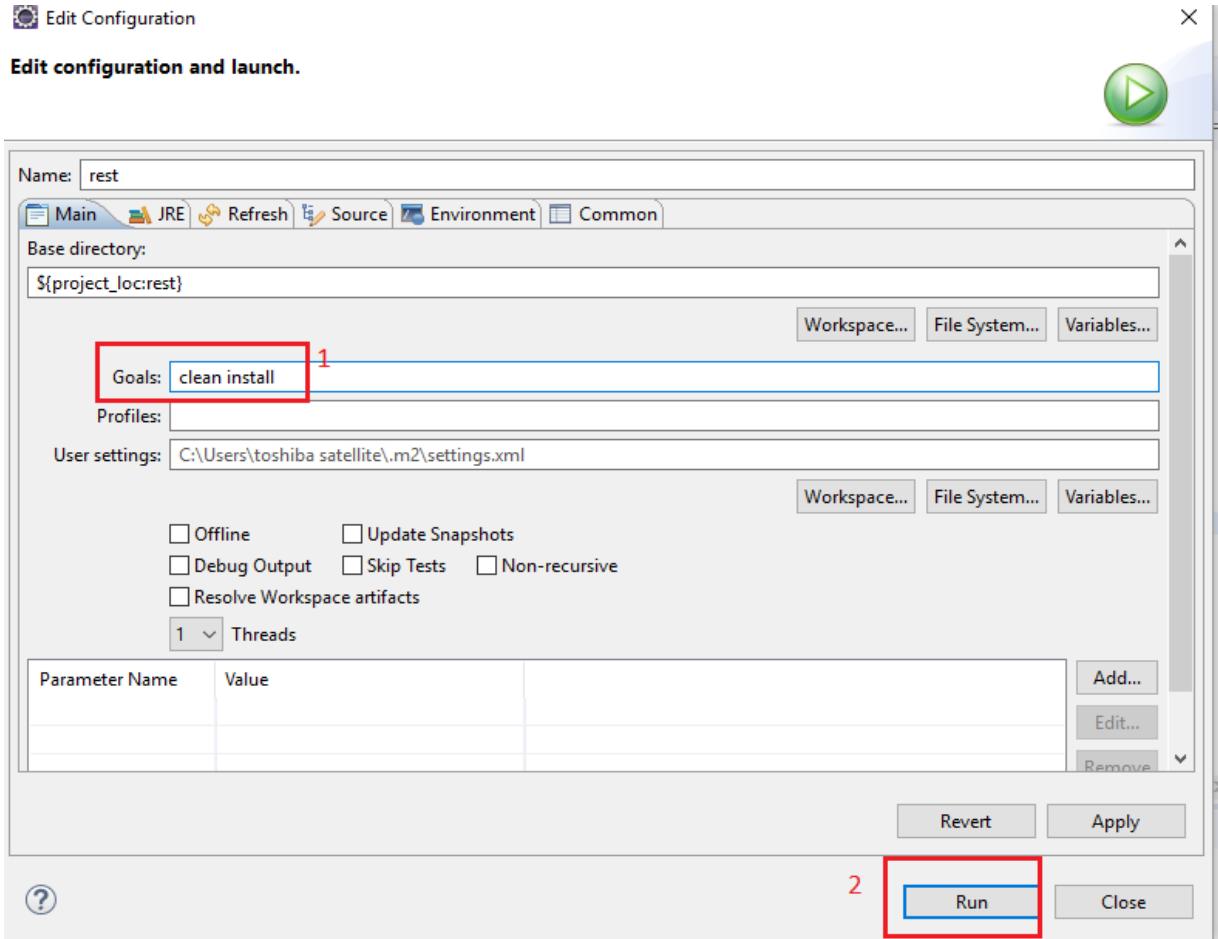
- @RestController est une annotation utilisée au niveau de la classe. L'annotation @RestController marque la classe en tant que contrôleur où chaque méthode renvoie un objet de domaine au lieu d'une vue, c'est une version spécialisée du contrôleur. Il inclut les annotations @Controller et @ResponseBody et simplifie donc la mise en œuvre du contrôleur.

6. Build de l'application

***Packager votre application (créer le fichier exécutable de votre application)** : Pour se faire, cliquer à droite de la souris sur le fichier pom.xml ou bien à droite de la souris de votre projet :



- Cliquer sur « Run As » et ensuite sur « Maven build ... », la fenêtre suivante sera affichée :



- 1 : Saisir dans Goals : ***clean install***.
- 2 : Cliquer sur Run. Vérifier que le build de votre application a été bien effectuée sans erreur :

```

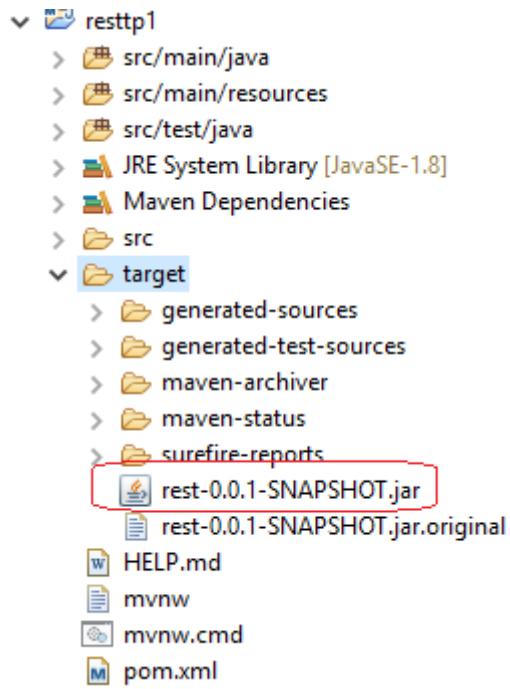
<terminated> rest [Maven Build] C:\Program Files\Java\jdk1.8.0_121\bin\javaw.exe (3 mai 2019 à 23:14:53)

.
.
.
:: Spring Boot :: (v2.2.0.BUILD-SNAPSHOT)

2019-05-03 23:15:06.481 INFO 5812 --- [           main] ma.cigma.rest.RestApplicationTests    : Starting RestApplicationTests on hassane with PID 5812 (started by toshi
2019-05-03 23:15:06.485 INFO 5812 --- [           main] ma.cigma.rest.RestApplicationTests    : No active profile set, falling back to default profiles: default
2019-05-03 23:15:08.588 INFO 5812 --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2019-05-03 23:15:09.060 INFO 5812 --- [           main] ma.cigma.rest.RestApplicationTests    : Started RestApplicationTests in 3.052 seconds (JVM running for 4.388)
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 4.183 s - in ma.cigma.rest.RestApplicationTests
2019-05-03 23:15:09.634 INFO 5812 --- [           Thread-2] o.s.s.concurrent.ThreadPoolTaskExecutor : Shutting down ExecutorService 'applicationTaskExecutor'
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
[INFO] --- maven-jar-plugin:3.1.1:jar (default-jar) @ rest ---
[INFO] Building jar: C:\webservices\rest\target\rest-0.0.1-SNAPSHOT.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:2.2.0.BUILD-SNAPSHOT:repackage (repackage) @ rest ---
[INFO] Replacing main artifact with repackaged archive
[INFO]
[INFO] --- maven-install-plugin:2.5.2:install (default-install) @ rest ---
[INFO] Installing C:\webservices\rest\target\rest-0.0.1-SNAPSHOT.jar to C:\Users\toshiba satellite\.m2\repository\ma\cigma\rest\0.0.1-SNAPSHOT\rest-0.0.1-SNAPSHOT.jar
[INFO] Installing C:\webservices\rest\pom.xml to C:\Users\toshiba satellite\.m2\repository\ma\cigma\rest\0.0.1-SNAPSHOT\rest-0.0.1-SNAPSHOT.pom
[INFO]
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 17.872 s
[INFO] Finished at: 2019-05-03T23:15:13+01:00
[INFO] Final Memory: 38M/216M
[INFO] -----

```

- Vérifier également que le fichier \rest\target\rest-* .jar a été bien créé :



- Pour exécuter votre application, lancer la méthode main de la classe MainApplication ou bien, lancer l'invite de commande et lancer la commande suivante :

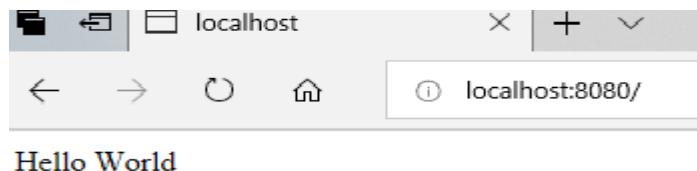
java -jar C:\webservices\rest\target\rest-0.0.1-SNAPSHOT.jar comme illustré ci-après :

```
C:\webservices\rest\target>java -jar rest-0.0.1-SNAPSHOT.jar
```

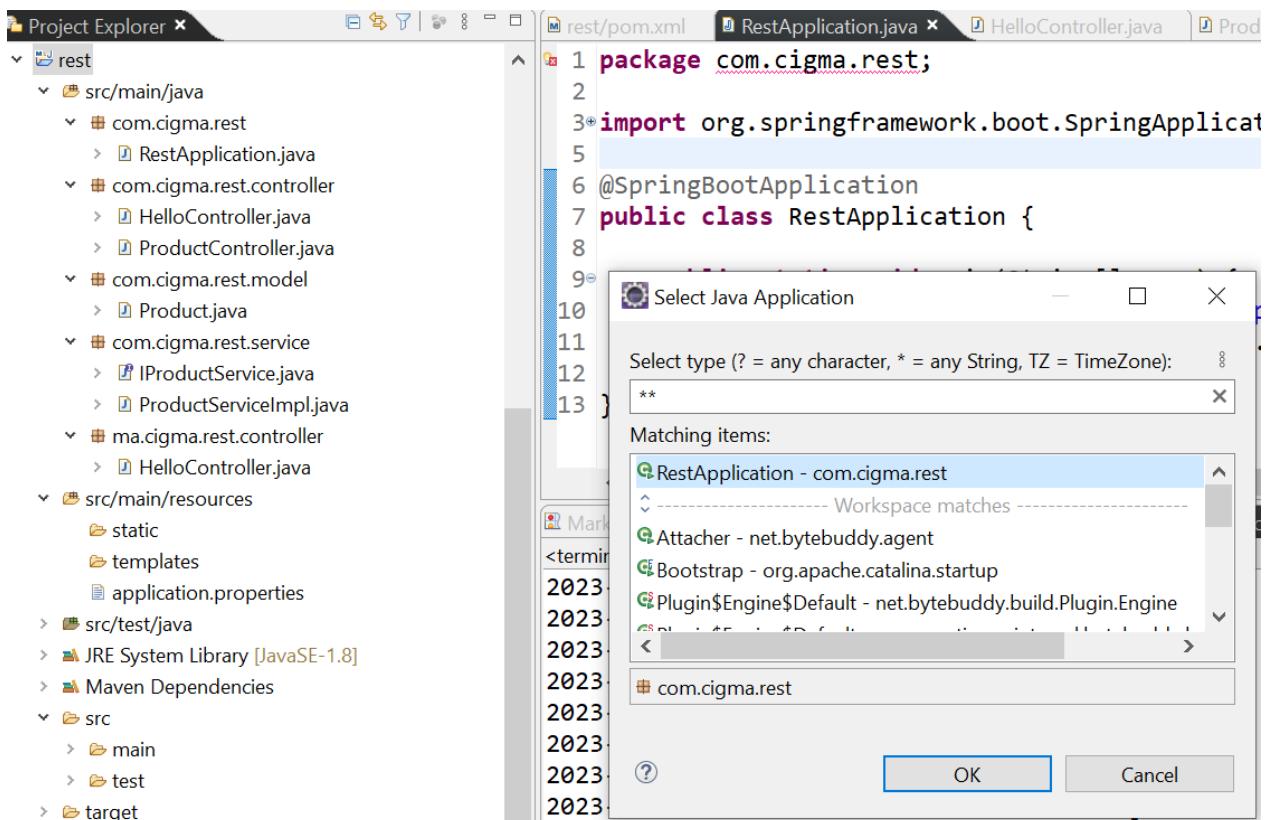
```
2019-05-05 12:48:36.896 INFO 1100 --- [           main] ma.cigma.rest.MainApplication        : Starting MainApplication on hassane with PID 1100 (C:\webservices\resttp1\target\)
2019-05-05 12:48:36.901 INFO 1100 --- [           main] ma.cigma.rest.MainApplication        : No active profile set, falling back to default profiles: default
2019-05-05 12:48:38.619 INFO 1100 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer: Tomcat initialized with port(s): 8080 [http]
2019-05-05 12:48:38.660 INFO 1100 --- [           main] o.apache.catalina.core.StandardService: Starting service [Tomcat]
2019-05-05 12:48:38.661 INFO 1100 --- [           main] org.apache.catalina.core.StandardEngine: Starting Servlet engine: [Apache Tomcat/9.0.19]
2019-05-05 12:48:38.661 INFO 1100 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]: Initializing Spring embedded WebApplicationContext
2019-05-05 12:48:38.898 INFO 1100 --- [           main] o.s.web.context.ContextLoader        : Root WebApplicationContext: initialization completed in 1905 ms
2019-05-05 12:48:38.898 INFO 1100 --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor: Initializing ExecutorService 'applicationTaskExecutor'
2019-05-05 12:48:39.456 INFO 1100 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer: Tomcat started on port(s): 8080 (http) with context path ''
2019-05-05 12:48:39.460 INFO 1100 --- [           main] ma.cigma.rest.MainApplication        : Started MainApplication in 3.089 seconds (JVM running for 3.632)
2019-05-05 12:48:44.305 INFO 1100 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]: Initializing Spring DispatcherServlet 'dispatcherServlet'
2019-05-05 12:48:44.305 INFO 1100 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet     : Initializing Servlet 'dispatcherServlet'
2019-05-05 12:48:44.315 INFO 1100 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet     : Completed initialization in 10 ms
```

NB : Observez au niveau de la console que Spring Boot démarre Tomcat (Embedded Server Runtime) au port 8080. Nous reviendrons vers ce point à la suite du TP et nous allons voir comment modifier ce port.

- Pour tester votre application, lancer le lien <http://localhost:8080> ou bien <http://localhost:8080/hello>, le résultat est :



Ou bien lancer l'exécution à partir de Eclipse :



7. Développement du contrôleur ProductController

Maintenant nous allons développer le contrôleur ProductController qui offrira les services CRUD. Dans notre exemple, nous allons utiliser une liste statique des Produits.

- Commençons tout d'abord par créer le modèle (la classe `ma.cigma.rest.model.Product`):

```
package ma.cigma.rest.service.model;

public class Product {
    private Long id;
    private String name;

    public Product() {
    }
    public Product(Long id, String name) {
        this.id = id;
        this.name = name;
    }
    public Long getId() {
        return id;
    }
}
```

```

    public void setId(Long id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}

```

- Créer ensuite l'interface IProductService et la classe ProductServiceImpl :

```

package ma.cigma.rest.service;

import java.util.List;

import ma.cigma.rest.service.model.Product;

public interface IProdcutService {
    Product getById(Long id);
    List<Product> getAll();
    void create(Product product);
    void update(Long id, Product product);
    void delete(Long id);
}

```

```

package ma.cigma.rest.service;

import java.util.ArrayList;
import java.util.List;

import org.springframework.stereotype.Service;

import ma.cigma.rest.service.model.Product;

/**
 * Il est recommandé d'annoter les classes de la couche métier par @Service
 * Spring injectera par la suite un objet de cette classe au niveau du contrôleur.
 */
@Service
public class ProductServiceImpl implements IProdcutService {
    private static List<Product> productRepo = new ArrayList<>();
    static {
        productRepo.add(new Product(11, "PC PORTABLE HP"));
        productRepo.add(new Product(21, "TV LG 32p"));
        productRepo.add(new Product(31, "TV Sony 49p"));
        productRepo.add(new Product(41, "Camera Sony"));
    }

    @Override
    public Product getById(Long id) {
        if (productRepo == null || productRepo.isEmpty())
            return null;
        for (Product product : productRepo) {
            if (id.equals(product.getId()))

```

```

        return product;
    }
    return null;
}

@Override
public List<Product> getAll() {
    return productRepo;
}

@Override
public void update(Long id, Product product) {
    Product productFound = getById(id);
    if (productFound == null)
        return;
    productRepo.remove(productFound);
    product.setId(id);
    productRepo.add(product);
}

@Override
public void delete(Long id) {
    Product productFound = getById(id);
    if (productFound == null)
        return;
    productRepo.remove(productFound);
}

@Override
public void create(Product product) {
    productRepo.add(product);
}
}

```

- Créer la classe ProductController :

```

package ma.cigma.rest.controller;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

import ma.cigma.rest.service.IProdcutService;
import ma.cigma.rest.service.model.Product;

@RestController
public class ProductController {

```

```

/**
 * @Autowired permet d'injecter le bean de type IProdcutService
 * (objet représentant la couche métier).
 * Ici, le Design Pattern qui est appliqué est l'IOC (Inversion Of Control).
 */
@.Autowired
private IProdcutService service;

/**
 * Pour chercher tous les produits
 */
@GetMapping(value = "/products")
public List<Product> getAll() {
    return service.getAll();
}

/**
 * Pour chercher un produit par son id
 */
@GetMapping(value = "/products/{id}")
public Product getProductById(@PathVariable(name = "id") Long productId) {
    return service.getById(productId);
}

/**
 * Pour créer un nouveau produit
ResponseEntity représente l'intégralité de la réponse HTTP : code d'état, en-têtes et corps . Par conséquent, nous pouvons l'utiliser pour configurer entièrement la réponse HTTP. Si nous voulons l'utiliser, nous devons la renvoyer depuis le point de terminaison ; Spring s'occupe du reste.
*/
@PostMapping(value = "/products")
public ResponseEntity<Object> createProdut(@Validated @RequestBody Product product) {
    service.create(product);
    return new ResponseEntity<>("Product is created successfully",
HttpStatus.CREATED);
}

/**
 * Pour modifier un produit par son id
*/
@PutMapping(value = "/products/{id}")
public ResponseEntity<Object> updateProduct(@PathVariable(name = "id") Long productId,
                                              @RequestBody Product product) {
    Product productFound = service.getById(productId);
    if (productFound == null)
        return ResponseEntity.notFound().build();
    service.update(productId, product);
    return new ResponseEntity<>("Product is updated successsfully",
HttpStatus.OK);
}

/**
 * Pour supprimer un produit par son id
*/
@DeleteMapping(value = "/products/{id}")
public ResponseEntity<Object> deleteProduct(@PathVariable(name = "id") Long productId) {
    Product productFound = service.getById(productId);

```

```

        if (productFound == null)
            return ResponseEntity.notFound().build();
        service.delete(productId);
        return new ResponseEntity<>("Product is deleted successfully",
HttpStatus.OK);
    }

    public IProdcutService getService() {
        return service;
    }

    public void setService(IProdcutService service) {
        this.service = service;
    }
}

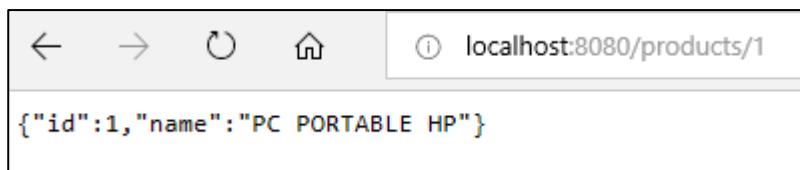
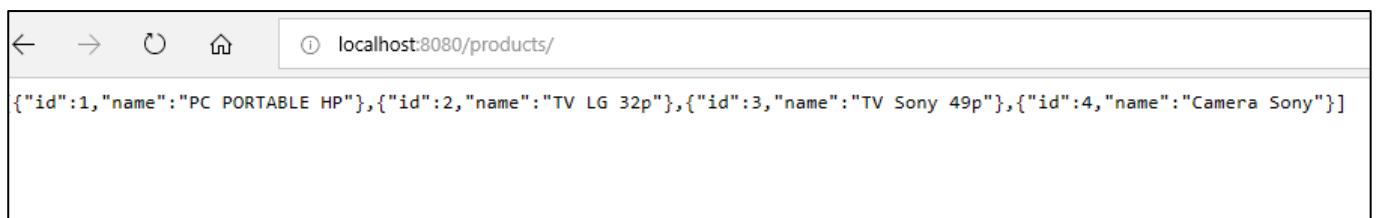
```

8. Rebuild de l'application

Refaire les mêmes étapes définies dans le chapitre 6 ci-dessus.

9. Tester les services Rest de ProductController

- Pour tester les méthode GET, il suffit de lancer le lien : <http://localhost:8080/products> pour chercher la liste des produits et <http://localhost:8080/products/1> pour chercher le produit dont l'identifiant est 1. Les résultats devront être :



- Pour tester les méthodes PUT, POST et DELETE, utiliser l'outil POSTMAN :
 - o Tout d'abord, Afficher le résultat du GET :

GET http://localhost:8080/products/1

Params Authorization Headers (7) Body Pre-request Script Tests Settings

Query Params

	Key	Value	Description
	Key	Value	Description

Body Cookies Headers (5) Test Results 200 OK 13 ms 196 B

Pretty Raw Preview Visualize JSON

```

1 [
2   "id": 1,
3   "name": "PC PORTABLE HP"
4 ]

```

- Pour PUT :

PUT http://localhost:8080/products/1

No Environment

http://localhost:8080/products/1

PUT http://localhost:8080/products/1 Send Save

Params Authorization Headers (9) Body Body (green) Pre-request Script Tests Cookies Code Comment

none form-data x-www-form-urlencoded raw binary JSON (application/json)

{"name": "PC PORTABLE DELL"}

1 2 3 4 5 6 7

Status: 200 OK Time: 6 ms Size: 148 B Download

Product is updated successfully Le résultat s'affiche ici

- 1 : Entrer l'URL : http://localhost:8080/products/1
- 2: Choisir PUT
- 3 : Cliquer sur Body
- 4 : Cliquer sur raw
- 5 : Entrer le message JSON : {"name":"PC PORTABLE DELL"}
- 6 : Sélectionner JSON(application/json)
- 7 : Cliquer sur Send et observer le résultat.

- Pour POST :

The screenshot shows the Postman interface with the following details:

- Method:** POST (2)
- URL:** http://localhost:8080/products/1 (1)
- Body tab selected (3):** Body type is raw (4), content is JSON (application/json) (6), and the message body is {"id":5,"name":"Nouveau produit"} (5).
- Send button (7):** The "Send" button is highlighted.
- Status bar:** Status: 201 Created, Time: 6 ms, Size: 152 B.
- Message:** Product is created successfully (1).

1 : Entrer l'URL : <http://localhost:8080/products>

2: Choisir POST

3 : Cliquer sur Body

4 : Cliquer sur raw

5 : Entrer le message JSON : {"id":5,"name":"Nouveau produit"}

6 : Sélectionner JSON(application/json)

7 : Cliquer sur Send et observer le résultat.

The screenshot shows the Postman interface with the following details:

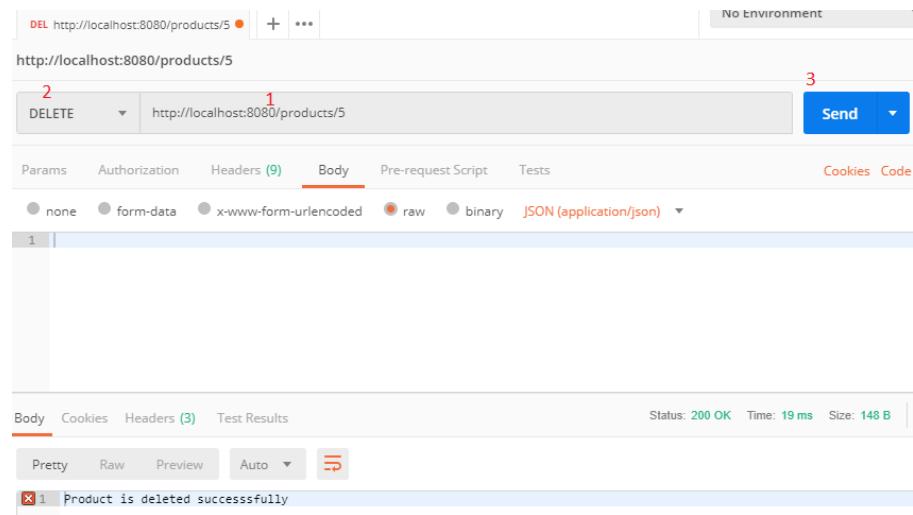
- Method:** GET
- URL:** http://localhost:8080/products
- Body tab selected:** Body type is raw, content is JSON, and the message body is [{"id":5,"name":"My Nouveau produit"}].
- Status bar:** Status: 200 OK, Time: 4 ms, Size: 325.
- Response body (Pretty view):**

```

9   [
10    {
11      "id": 4,
12      "name": "Camera Sony"
13    },
14    {
15      "id": 1,
16      "name": "PC PORTABLE DELL"
17    },
18    {
19      "id": 5,
20      "name": "My Nouveau produit"
21    }
22 ]

```

○ Pour DELETE :



1 : Entrer l'URL : <http://localhost:8080/products/5>

2: Choisir DELETE

3 : Cliquer sur Send et observer le résultat.

10. Produire le format XML

Maintenant nous allons voir comment notre contrôleur puisse produire le format XML.

La première chose qu'il faut faire est d'ajouter la dépendance suivante au niveau de pom.xml :

```
<dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

Ensuite, ajouter « produces={MediaType.APPLICATION_XML_VALUE, MediaType.APPLICATION_JSON_VALUE} » au niveau de vos annotations @GetMapping, @PostMapping, ... :

Exemple :

```
@GetMapping(value =
"/products", produces={MediaType.APPLICATION_XML_VALUE, MediaType.APPLICATION_JSON_VALUE})
public List<Product> getAll() {
    return service.getAll();
}
```

Pour tester le format XML, préciser dans le flag Accept de votre Header la valeur « application/xml ». Voir imprimé écran ci-après :

GET http://localhost:8080/products/ + ...

NO Environment

GET http://localhost:8080/products/ Send

Params Authorization Headers (7) Body Pre-request Script Tests Cookies Code

Headers (1)

KEY	VALUE	DESCRIPTION	...	Bulk Edit
<input checked="" type="checkbox"/> Accept	application/xml			
Key	Value	Description		

Temporary Headers (6) ①

Body Cookies Headers (3) Test Results Status: 200 OK Time: 4 ms Size: 331 B

Pretty Raw Preview XML

```

1 <List>
2   <item>
3     <id>1</id>
4     <name>PC PORTABLE HP</name>
5   </item>
6   <item>
7     <id>2</id>
8     <name>TV LG 32p</name>
9   </item>
10  </items>

```

11. Les fichiers de configuration de l'application de Spring Boot : application-X.properties

Vous remarquez que le fichier **application.properties** existe au niveau du chemin src/main/resources.

Spring offre la possibilité de créer un fichier par environnement. Par exemple, vous pouvez créer le fichier **application-prod.properties** pour votre environnement de production, **application.properties** pour votre environnement de développement et **application-integration.properties** pour votre environnement d'intégration. Le fichier par défaut est **application.properties**. Ici on parle de la notion **SPRING ACTIVATE PROFILE**.

Par défaut, Spring Boot démarre Tomcat au port 8080. Vous pouvez bien sûr changer ce port. Par exemple, sur l'environnement de production, nous allons démarrer Tomcat au port 4431, sur l'intégration, nous allons démarrer Tomcat au port 9090 et finalement sur l'environnement de développement nous allons utiliser le port par défaut qui est 8080.

- Modifier le fichier **application.properties** comme suit :

```
server.port=8080
spring.application.name=tp1_springroot_restfull
```

- Créer le fichier **application-prod.properties** comme suit :

```
server.port=4431
spring.application.name=tp1_environnement_de_Prodcution
```

- Créer le fichier ***application-integration.properties*** comme suit :

```
server.port=8081  
spring.application.name=tp1_environnement_integration
```

- Pour utiliser le fichier ***application-prod.properties***, lancer la commande suivante :

```
java -jar C:\webservices\resttp1\target\rest-0.0.1-SNAPSHOT.jar  
--spring.profiles.active=prod
```

Observer que Spring Boot a bien démarré Tomcat au port 4431.

- De même, pour utiliser le fichier ***application-integration.properties***, lancer la commande suivante :

```
java -jar C:\webservices\resttp1\target\rest-0.0.1-SNAPSHOT.jar  
--spring.profiles.active=integration
```

Observer que Spring Boot démarre maintenant Tomcat au port 9090.

12. Lire une valeur à partir du fichier application-X.properties

Maintenant nous allons voir comment Spring Boot puisse récupérer une valeur d'une clé se trouvant au niveau du fichier application-X.properties.

Spring Boot fourni l'annotation @Value. Voir le code de la classe HelloController2 :

```
package ma.cigma.rest.controller;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloController2 {
    @Value("${spring.application.name}")
    private String name;

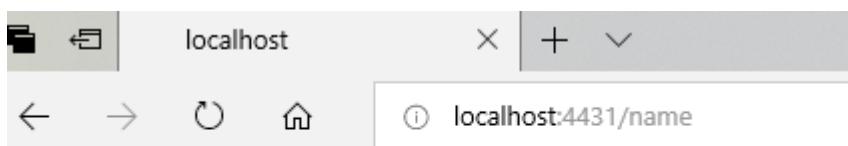
    @RequestMapping(value = "/name")
    public String name() {
        return name;
    }
}
```

Rafaire le Build de votre application puis démarrer et ensuite exécuter la commande :

```
java -jar C:\webservices\resttp1\target\rest-0.0.1-SNAPSHOT.jar --spring.profiles.active=prod
```

et lancer ensuite le lien : <http://localhost:4431/name>

Le résultat devrait être :



nom_application_environnement_de_Prodction

13. Utiliser un fichier de configuration externe

- Copier par exemple votre fichier application-prod.properties dans c:\webservices et modifier le port et le nom de votre application. Exemple :

```
server.port=7777
spring.application.name=nom_application_environnement_de_Prodction(A partir d'un fichier externe)
```

- Pour que Spring Boot utilise ce fichier pour démarrer l'application, lancer la commande suivante :

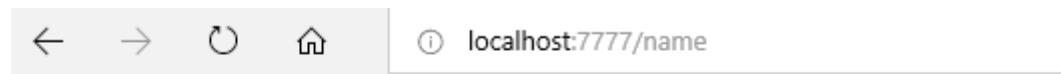
```
java -jar -Dspring.config.location=C:\webservices\application-prod.properties  
C:\webservices\resttp1\target\rest-0.0.1-SNAPSHOT.jar
```

Observer le flag **-Dspring.config.location**.

Le résultat devrait être :

- Lancer le lien : <http://localhost:7777/name>

Le résultat devrait être :



nom_application_environnement_de_Prodction(A partir d'un fichier externe)