

## Création d'un microservice avec Spring Boot

Vous avez précédemment développé des services Rest en utilisant Jersey. Vous avez mis en place toute la configuration nécessaire pour votre projet. Via ce tuto, vous allez découvrir le framework Spring Boot.

Spring Boot est un framework dédié au développement d'applications Java. Il offre des facilités telle que l'auto-configuration. Contrairement à son prédécesseur Spring MVC, Spring Boot vous permet de développer vos applications sans vous plonger dans les fichiers XML de configuration.

### a. Questions :

- C'est quoi un microservice?
- Quelles sont les caractéristiques des microservices?

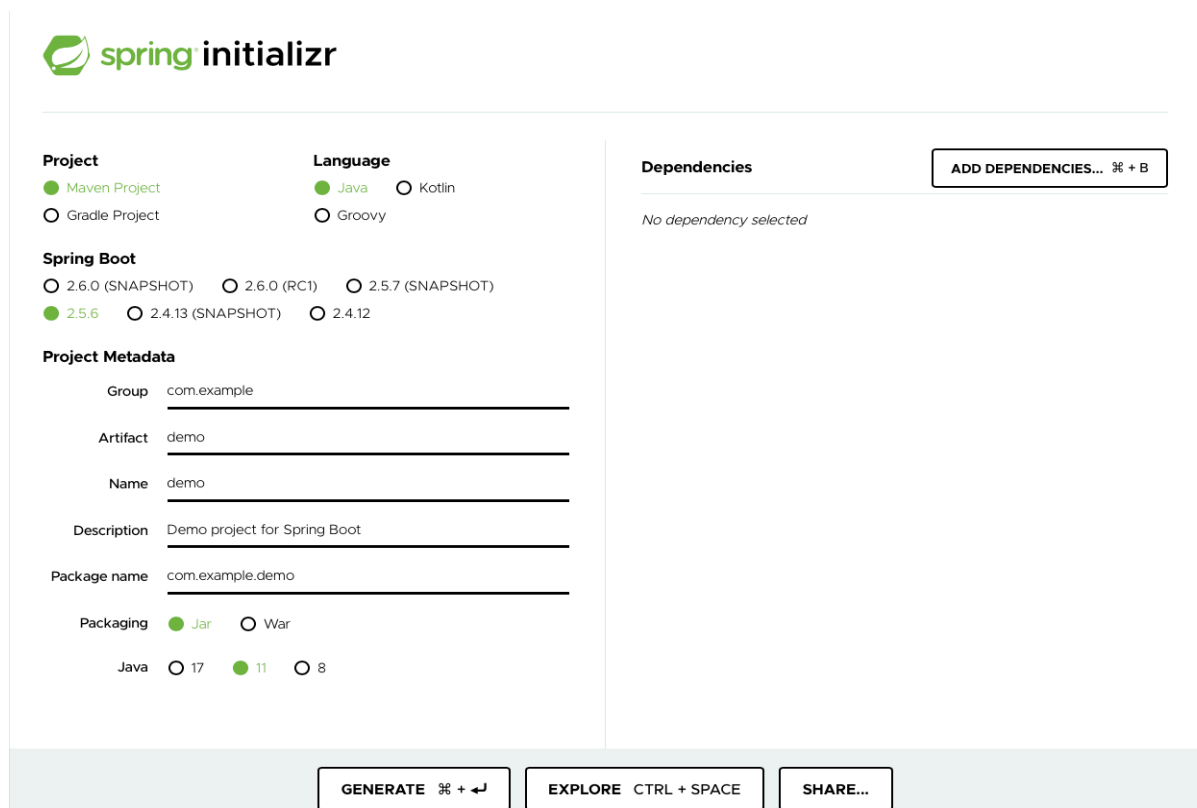
### b. Compétences visées :

Le but ici est de créer un microservice qui expose une API Rest et ce en utilisant Spring Boot.

### c. Créer votre microservice

Vous allez créer un microservice qui gère des étudiants. Allez sur le site de Spring Boot <https://start.spring.io>

Via ce site, vous allez pouvoir créer un projet Spring Boot.



The screenshot shows the Spring Initializr web interface. The 'Project' section has 'Maven Project' selected. The 'Language' section has 'Java' selected. The 'Spring Boot' section has '2.5.6' selected. The 'Project Metadata' section contains fields for Group (com.example), Artifact (demo), Name (demo), Description (Demo project for Spring Boot), and Package name (com.example.demo). The 'Packaging' section has 'Jar' selected. The 'Dependencies' section is empty with a button to 'ADD DEPENDENCIES...'. At the bottom, there are buttons for 'GENERATE', 'EXPLORE', and 'SHARE...'.

Project	Language	Spring Boot	Project Metadata	Packaging	Dependencies
<input checked="" type="radio"/> Maven Project <input type="radio"/> Gradle Project	<input checked="" type="radio"/> Java <input type="radio"/> Kotlin <input type="radio"/> Groovy	<input type="radio"/> 2.6.0 (SNAPSHOT) <input type="radio"/> 2.6.0 (RC1) <input type="radio"/> 2.5.7 (SNAPSHOT) <input checked="" type="radio"/> 2.5.6 <input type="radio"/> 2.4.13 (SNAPSHOT) <input type="radio"/> 2.4.12	Group: com.example Artifact: demo Name: demo Description: Demo project for Spring Boot Package name: com.example.demo	<input checked="" type="radio"/> Jar <input type="radio"/> War Java: <input type="radio"/> 17 <input checked="" type="radio"/> 11 <input type="radio"/> 8	<input type="button" value="ADD DEPENDENCIES..."/> No dependency selected

Buttons: GENERATE, EXPLORE, SHARE...

Il suffit de renseigner les différents champs comme suit :

The screenshot shows a web form for configuring a Spring Boot project. It is divided into several sections:

- Project:** Radio buttons for **Maven Project** (selected) and **Gradle Project**.
- Language:** Radio buttons for **Java** (selected), **Kotlin**, and **Groovy**.
- Spring Boot:** Radio buttons for versions: **2.6.0 (SNAPSHOT)**, **2.6.0 (RC1)**, **2.5.7 (SNAPSHOT)**, **2.5.6** (selected), **2.4.13 (SNAPSHOT)**, and **2.4.12**.
- Project Metadata:** A series of text input fields:
  - Group:** `fr.insa.mas`
  - Artifact:** `studentManagementMS`
  - Name:** `studentManagementMS`
  - Description:** `Example of MS`
  - Package name:** `fr.insa.mas.studentManagementMS`
- Packaging:** Radio buttons for **Jar** (selected) and **War**.
- Java:** Radio buttons for versions **17**, **11** (selected), and **8**.

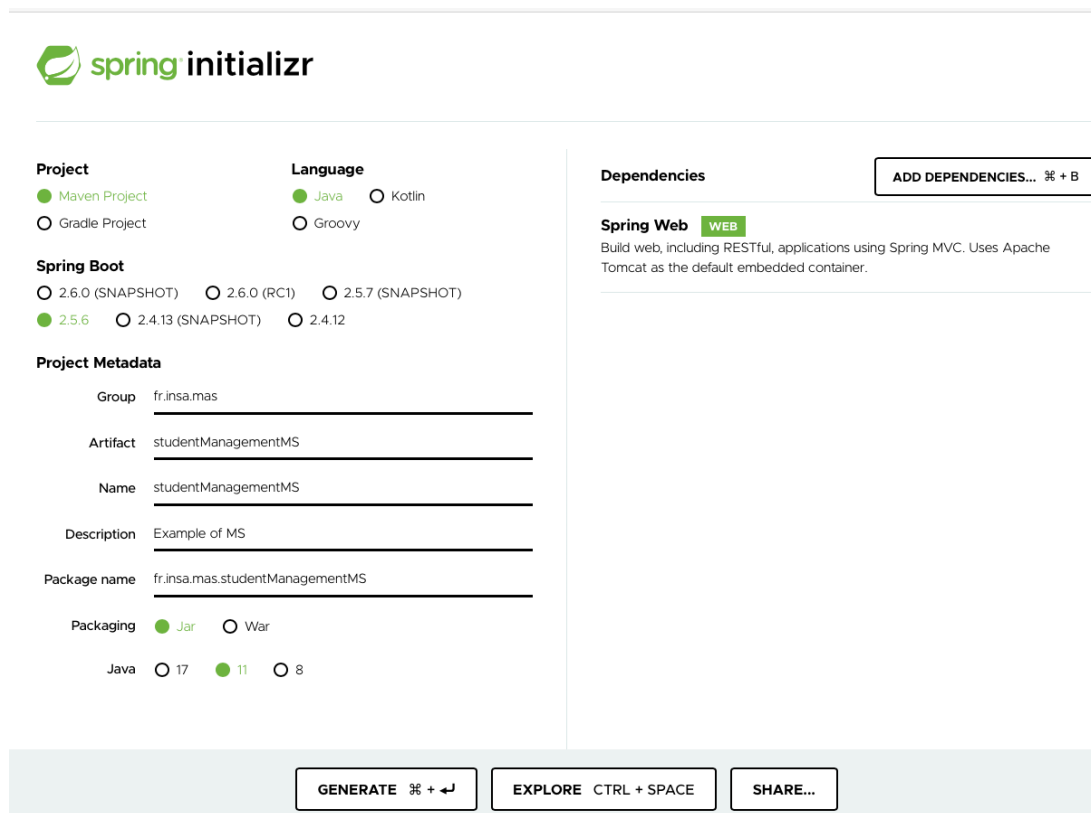
A droite, vous remarquerez la section « Dependencies ». Vous pouvez ajouter des dépendances dont vous avez besoin dans votre projet. Spring Boot se charge ensuite de les importer et de les configurer pour vous.

Pour le projet que vous allez développer, vous auriez besoin d'ajouter des dépendances pour le développement d'applications Web, de mettre en place une API Rest,...etc. Pour cela, cliquez sur « Add dependencies » puis chercher le mot clé « Web » comme montré ci-dessous et choisir « Spring Web » :

The screenshot shows the search results for the keyword "web" in the Spring Boot dependency manager. At the top, it says "web" and "Press ⌘ for multiple adds". The results are listed as follows:

- Spring Web** (tag: WEB): Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
- Spring Reactive Web** (tag: WEB): Build reactive web applications with Spring WebFlux and Netty.
- Thymeleaf** (tag: TEMPLATE ENGINES): A modern server-side Java template engine for both web and standalone environments. Allows HTML to be correctly displayed in browsers and as static prototypes.
- Spring Web Services** (tag: WEB): Facilitates contract-first SOAP development. Allows for the creation of flexible web services using one of the many ways to manipulate XML payloads.
- WebSocket** (tag: MESSAGING): Build WebSocket applications with SockJS and STOMP.
- Jersey** (tag: WEB): Framework for developing RESTful Web Services in Java that provides support for JAX-RS APIs.

Maintenant votre projet est prêt.

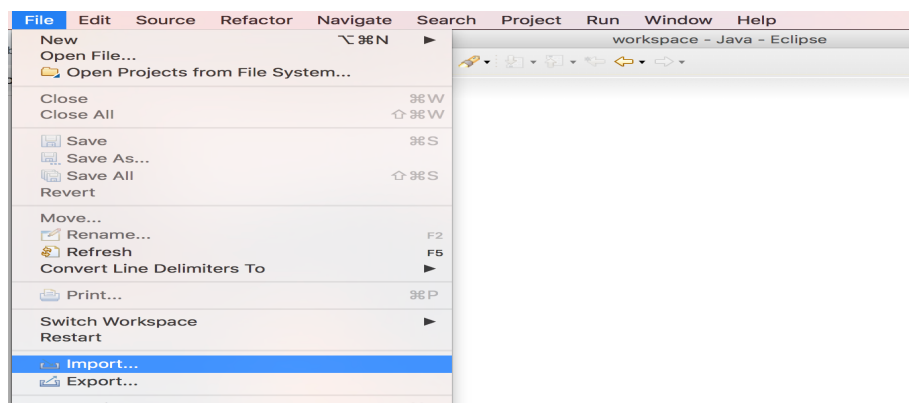


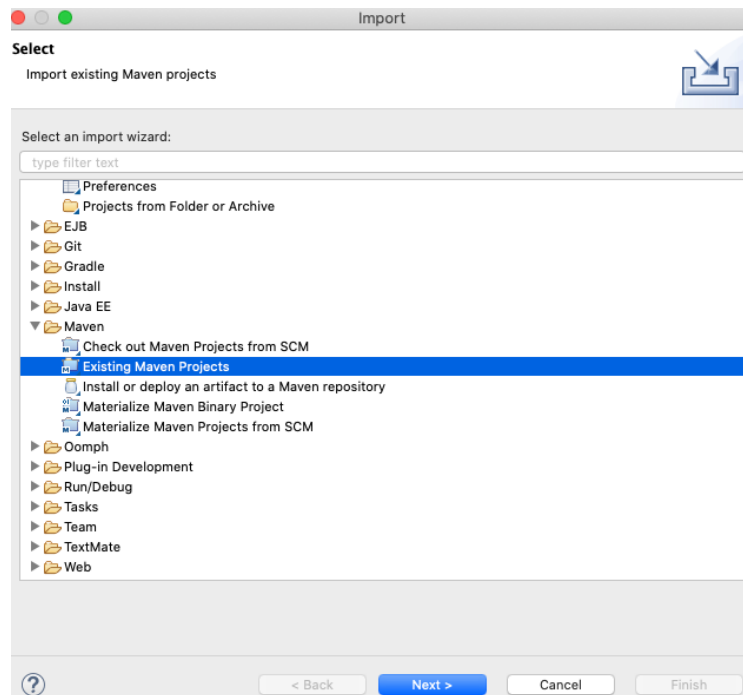
The image shows the Spring Initializr web form. It is divided into several sections: Project, Language, Spring Boot, Project Metadata, Dependencies, and a bottom bar with action buttons.

- Project:** ☒ Maven Project, ☐ Gradle Project
- Language:** ☒ Java, ☐ Kotlin, ☐ Groovy
- Spring Boot:** ☐ 2.6.0 (SNAPSHOT), ☐ 2.6.0 (RC1), ☐ 2.5.7 (SNAPSHOT), ☒ 2.5.6, ☐ 2.4.13 (SNAPSHOT), ☐ 2.4.12
- Project Metadata:**
  - Group:
  - Artifact:
  - Name:
  - Description:
  - Package name:
  - Packaging: ☒ Jar, ☐ War
  - Java: ☐ 17, ☒ 11, ☐ 8
- Dependencies:** . Below it, "Spring Web" is selected with a "WEB" tag. Description: "Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container."
- Bottom Bar:** , ,

Cliquez sur “Generate” pour télécharger le projet. Décompressez le fichier téléchargé.

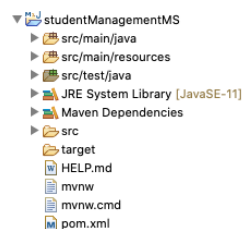
Allez sur Eclipse et importer le projet décompressé (File/Import/Maven/Existing Maven Projects):





Ensuite, il vous suffit d'indiquer le chemin vers le projet décompressé puis terminez.

Examinez l'arborescence du projet que vous venez de créer :

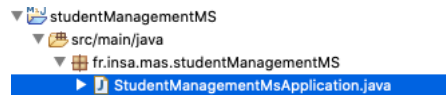


Comme vous pouvez le voir, Spring importe tous les .jar nécessaires pour votre projet. Vous pouvez les voir sous Maven Dependencies (comme vous l'avez certainement remarqué, il utilise Maven).



Sous *resources*, le fichier **application.properties** vous permettra de mettre en place certaines configurations. Par exemple, pour modifier le port de votre serveur de déploiement, il suffit d'ajouter la propriété suivante : **server.port 8081**

Spring génère sous `src/main/java` la class **StudentManagementMSApplication** dans le package *fr.insa.mas.studentManagementMS*.



Cette classe est la classe principale de Spring. Son rôle est de lancer l'application.

```
package fr.insa.mas.studentManagementMS;

import org.springframework.boot.SpringApplication;

@SpringBootApplication
public class StudentManagementMsApplication {

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

Spring Boot utilise les annotations. L'annotation **@SpringBootApplication** permet de mettre en place certaines étapes (comme par exemple l'auto-configuration) pour lancer votre application.

Avant de se lancer dans le code de votre microservice, commencez d'abord par exécuter la classe *StudentManagementMSApplication*. Exécutez là comme une application (c'est une classe qui a la méthode *main* !). Clic droit sur la classe *StudentManagementMSApplication* puis **Run AS/Java Application** :



Vous remarquerez depuis la console le lancement (déploiement) de votre projet via l'exécution de l'application principale de Spring. Le serveur tomcat écoute sur le port 8080.



```
2021-11-03 07:00:11.680 INFO 61718 --- [main] f.i.m.s.StudentManagementMsApplication : Starting StudentManagementMsApplication using Java 11.0.13 (
2021-11-03 07:00:11.682 INFO 61718 --- [main] f.i.m.s.StudentManagementMsApplication : No active profile set, falling back to default profiles: de
2021-11-03 07:00:12.728 INFO 61718 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2021-11-03 07:00:12.742 INFO 61718 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2021-11-03 07:00:12.743 INFO 61718 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.54]
2021-11-03 07:00:12.816 INFO 61718 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2021-11-03 07:00:12.817 INFO 61718 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 106
2021-11-03 07:00:13.210 INFO 61718 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
2021-11-03 07:00:13.220 INFO 61718 --- [main] f.i.m.s.StudentManagementMsApplication : Started StudentManagementMsApplication in 2.052 seconds (JVM
```

Allez sur [localhost:8080](http://localhost:8080) (si vous n'avez pas changé de port). Puisque l'application est vide, vous allez obtenir la page suivante. Cette page vous indique qu'il n'est pas possible d'afficher des erreurs. C'est normal, votre application n'est pas encore implémentée.

## Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Wed Nov 03 07:01:34 CET 2021

There was an unexpected error (type=Not Found, status=404).

Votre projet est prêt, vous pouvez commencer l'implémentation de votre microservice qui va gérer des étudiants. Votre microservice est le contrôleur. Créez la classe *StudentResource* dans le package dédié aux contrôleurs (ici le package est *fr.insa.mas.studentManagementMS.controller*).

Ajoutez l'annotation **@RestController**, afin que votre microservice soit le contrôleur qui recevra les requêtes Rest.

```
package fr.insa.mas.studentManagementMS.controller;

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

@RestController
public class StudentResource {

}
```

Ajoutez la méthode *studentNumber()* qui retourne le nombre d'étudiants. Cette méthode va être exposée en utilisant la méthode GET de la ressource students. Donc, ajoutez l'annotation **@GetMapping** comme suit :

```
package fr.insa.mas.studentManagementMS.controller;

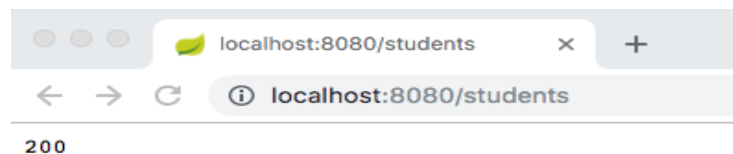
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class StudentResource {

    @GetMapping("/students")
    public int studentNumber(){
        return 200;
    }

}
```

Arrêtez l'application et relancer la. Testez votre microservice: [localhost:8080/students](http://localhost:8080/students)



Votre microservice fonctionne. Vous allez l'étendre de telle sorte que pour un id donné, il va retourner le nom, prénom de l'étudiant correspondant (simulation de récupération d'un étudiant d'une BDD).

Vous auriez besoin de définir la classe Student (i.e., le modèle). Créez la classe Student dans le package dédié au modèle (ici le package est *fr.insa.mas.studentManagementMS.model*).

Afin de générer le constructeur, les getters et les setters, allez dans eclipse/Source/Generate Getters and Setters and Generate Constructor using fields. Générez aussi un constructeur vide afin de permettre la sérialisation/désérialisation de vos objets Student.

```
package fr.insa.mas.studentManagementMS.model;

@XmlRootElement(name = "Student")
public class Student {
    private int id;
    private String lastName;
    private String firstName;

    public Student(int id, String lastName, String firstName){
        this.id=id;
        this.lastName=lastName;
        this.firstName=firstName;
    }

    public Student(){
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstNale(String firstName) {
        this.firstName = firstName;
    }
}
```

Retour au microservice. Ajoutez une méthode qui reçoit un entier (l'id d'un étudiant) et retourne un objet Student. Comme vous pouvez le voir ci-dessous, il faut ajouter l'annotation `@GetMapping(students/{id})` et l'annotation `@PathVariable` pour seulement accepter des entiers dans les appels de la ressource (`/students/{id}`).

```
package fr.insa.mas.studentManagementMS.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

import fr.insa.mas.studentManagementMS.model.Student;

@RestController
public class StudentResource {

    @GetMapping("/students")
    public int studentNumber(){
        return 200;
    }

    @GetMapping(value="/students/{id}")
    public Student infosStudent(@PathVariable int id){
        Student student=new Student(id,"Rosa","Parks");
        return student;
    }
}
```

Enregistrez et redémarrez votre application. Testez votre nouvelle méthode

**Remarque:** Pour un affichage plus lisible de vos documents JSON, vous pouvez ajouter par exemple l'extension *JSONView* de chrome. Par exemple, vous passerez de

```
{"id":20,"lastName":"Rosa","firstName":"Parks"}
```

À

```
{
  id: 20,
  lastName: "Rosa",
  firstName: "Parks"
}
```

#### d. XML comme format de données avec Spring :

Comme vous l'avez certainement remarqué, le format par défaut utilisé par Spring est JSON. Vous allez définir une nouvelle méthode qui produit des données en XML.

Ajoutez une nouvelle méthode à votre classe qui retourne un étudiant. Afin de spécifier que la nouvelle méthode retourne des données XML, il suffit d'ajouter l'annotation *produces* pour spécifier le format de données retourné.

Rappelez vous que pour la sérialisation de vos objets Student en XML, il faut ajouter l'annotation de JAXB `@XmlRootElement(name = "Student")` à votre modèle (i.e., à la classe Student).

Ajoutez dans le fichier pom.xml les dépendance suivantes pour importer les librairies nécessaires :

```
<dependency>
    <groupId>com.sun.xml.bind</groupId>
    <artifactId>jaxb-core</artifactId>
    <version>2.3.0.1</version>
</dependency>
<dependency>
    <groupId>javax.xml.bind</groupId>
    <artifactId>jaxb-api</artifactId>
</dependency>
<dependency>
    <groupId>com.sun.xml.bind</groupId>
    <artifactId>jaxb-impl</artifactId>
    <version>2.3.1</version>
</dependency>
<dependency>
    <groupId>org.javassist</groupId>
    <artifactId>javassist</artifactId>
    <version>3.25.0-GA</version>
</dependency>
```

N'oubliez pas de mettre à jour votre projet (clic droit sur le projet/Maven/Update Project ...).

L'annotation `@XmlRootElement(name = "Student")` doit être ajoutée avant la déclaration de votre classe.

```
package fr.insa.mas.studentManagementMS.Model;

import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name = "Student")
public class Student {
    private int id;
    private String lastName;
    private String firstName;

    public Student(int id, String lastName, String firstName) {
        this.id=id;
        this.lastName=lastName;
        this.firstName=firstName;
    }

    public Student() {}
}
```



Retournez à votre ressource. La méthode à ajouter est soulignée dans la figure suivante :

```
package fr.insa.mas.studentManagementMS.controller;

import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

import fr.insa.mas.studentManagementMS.model.Student;

@RestController
public class StudentResource {

    @GetMapping("/students")
    public int studentNumber(){
        return 200;
    }

    @GetMapping(value="/students/{id}")
    public Student infosStudent(@PathVariable int id){
        Student student=new Student(id,"Rosa","Parks");
        return student;
    }

    @GetMapping(value="/{id}", produces=MediaType.APPLICATION_XML_VALUE)
    public Student xmlInfosStudent(@PathVariable int id){
        Student student=new Student(id,"Rosa","Parks");
        return student;
    }
}
```

Testez votre microservice.

## Exercice :

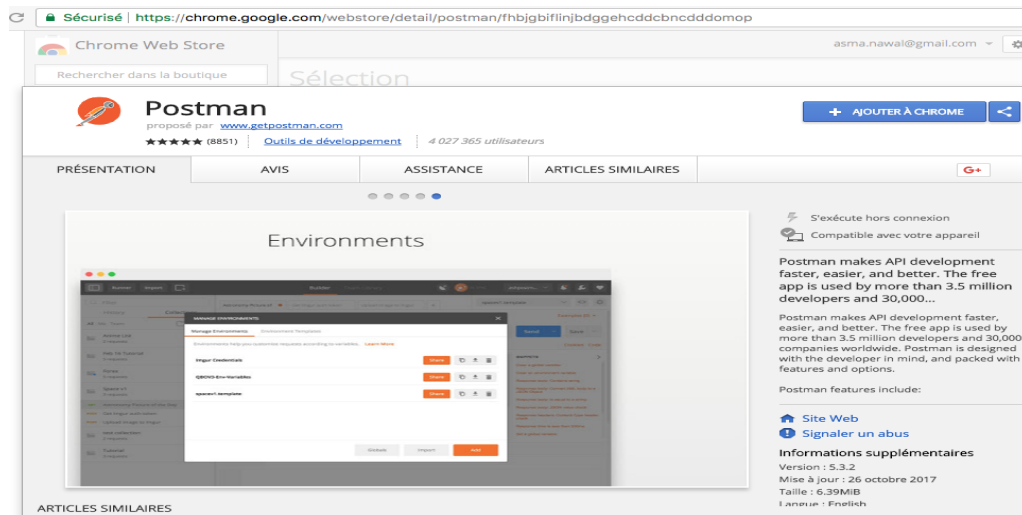
Étendez votre microservice pour permettre l'ajout de nouveaux étudiants, de mettre à jour les informations d'un étudiant existant. Vous pouvez simuler la BDD.

### Indications :

- Avec Spring Boot, d'une manière analogue à `@GetMapping`, les annotation suivantes existent : **`@PostMapping`**, **`@PutMapping`**, **`@DeleteMapping`**. Elles correspondent aux verbes http POST, PUT and DELETE.
- Comme vous l'avez constaté, l'annotation **`@PathVariable`** est utilisée pour des paramètres simples (int, float, ...). L'annotation **`@RequestBody`** est utilisée quand une méthode a comme paramètre d'entrée un objet.
- Dans le cas où une méthode a différents paramètres d'entrée, vous pouvez utiliser le code suivant :  
**`@GetMapping("/{RessourceName}/{param1}/{param2})`**  
**`methodName(@PathVariable param1, @PathVariable pam2)`**
- Pour tester des requêtes Rest, notamment Post, Put, et Delete, vous pouvez utiliser un client Rest, comme Postman de chrome.

### **Utilisation du client Rest Postman :**

Cherchez postman chrome sur google. Ajoutez-le.



Pour tester votre méthode PUT, utilisez Postman.

