

Les WEB Services de type REST

Généralités sur les services Web REST (RESTful web Services)

Un web service est une application serveur qui fournit des services sur le réseau Internet. Les services Web fournissent un moyen standard pour assurer l'interopérabilité d'applications qui s'exécutent sur des plateformes différentes et dans divers environnements.

Un Web service est un composant logiciel exécuté sur un serveur et accessible à travers le réseau.

Il y a 2 types de Web service :

- les services qui implémentent le protocole **SOAP** (Simple Object Access Protocol) qui utilise le format XML pour les messages échangés.
- les services de type **REST** (REpresentational State Transfer) basé sur le protocole HTTP.

Il existe en Java 2 implémentations principales des Web services REST qui fournissent toutes les fonctionnalités pour créer les services web **REST** (on dit aussi **RESTful**):

- L'écosystème Spring Boot qui inclut le framework **Spring Web**
- L'écosystème JEE qui inclut la librairie **Jersey** qui implémente l'API **JAX-RS**.

REST est basé sur le modèle d'architecture Client-Serveur.

Avec REST, les données et les fonctionnalités fournies par le serveur sont considérées comme des ressources.

Chaque ressource est identifiée par son URI et est accessible grâce à cet URI.

Les services web RESTful sur le Web sont sans état car ils reposent sur le protocole HTTP qui ne définit pas de session: chaque méthode HTTP s'exécute indépendamment de la précédente.

Les services web qui implémentent REST doivent fournir les opérations CRUD correspondantes aux méthodes HTTP:

- GET pour lire une ressource,
- POST pour créer une ressource, par exemple à partir de données d'un formulaire,
- PUT pour mettre à jour ou créer une ressource,
- DELETE pour supprimer une ressource.

Ces méthodes HTTP sont "envoyées" par le client. L'exécution de chacune de ces méthodes doit de plus renvoyer un code de retour vers le client.

La documentation Microsoft suivante sur les méthodes HTTP détaille les codes retournés au client pour chaque méthode.

Méthodes GET

En général, une méthode GET réussie renvoie le code d'état HTTP 200 (OK). Si la ressource est introuvable, la méthode doit renvoyer 404 (Introuvable).

Méthodes POST

Si une méthode POST crée une ressource, elle renvoie le code d'état HTTP 201 (Créé). L'URI de la nouvelle ressource est inclus dans l'en-tête **Location** de la réponse. Le corps de la réponse contient une représentation de la ressource.

Si la méthode effectue des opérations de traitement, mais ne crée pas de ressource, elle peut renvoyer le code d'état HTTP 200 et inclure le résultat de l'opération dans le corps de la réponse. Ou bien, en l'absence de résultat à renvoyer, la méthode peut renvoyer le code d'état HTTP 204 (Pas de contenu) sans corps de réponse.

Si le client place des données non valides dans la requête, le serveur doit renvoyer le code d'état HTTP 400 (Demande incorrecte). Le corps de la réponse peut contenir des informations supplémentaires sur l'erreur ou un lien vers un URI qui fournit plus de détails.

Méthodes PUT

Si une méthode PUT crée une ressource, elle renvoie le code d'état HTTP 201 (Créé), comme pour la méthode POST. Si la méthode met à jour une ressource existante, elle renvoie 200 (OK) ou 204 (Pas de contenu). Dans certains cas, mettre à jour une ressource existante peut se révéler impossible. Dans ce cas, envisagez de renvoyer le code d'état HTTP 409 (Conflit).

Méthodes DELETE

Si l'opération de suppression est réussie, le serveur web doit répondre avec un code d'état HTTP 204 indiquant que le processus a été géré correctement mais que le corps de la réponse ne contient aucune information supplémentaire. Si la ressource n'existe pas, le serveur web peut renvoyer HTTP 404 (Introuvable).

Un projet WebService REST

Créer un nouveau **Spring Starter Project**, donner lui un nom.

Déployer l'item **Web** et cocher uniquement **Spring Web** sur le formulaire des dépendances.

Spring Boot propose également le choix **Jersey** qui implémente l'API JAX-RS. La librairie **Jersey** est une autre solution pour créer des web services rest. L'écosystème JEE (Jakarta Enterprise Edition) utilise la librairie Jersey pour implémenter les web services rest.

► Exécuter le projet et observer les affichages dans la console.

The screenshot shows the 'New Spring Starter Project Dependencies' window. The 'Spring Boot Version' is set to 2.4.5. Under 'Frequently Used', 'Spring Web' is checked. In the 'Available' list, 'Web' is expanded and 'Spring Web' is checked. The 'Selected' list shows 'Spring Web'. At the bottom, the 'Finish' button is highlighted.


```

    }

    public void setNom(String nom) {
        this.nom = nom;
    }

    public String getPrenom() {
        return this.prenom;
    }

    public void setPrenom(String prenom) {
        this.prenom = prenom;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public Invite(long id, String nom, String prenom, String email) {
        this.id = id;
        this.nom = nom;
        this.prenom = prenom;
        this.email = email;
        LocalDate localdate = LocalDate.now();
        date = new Date(1000 * 24 * 3600 * localdate.toEpochDay());
        System.out.println("date de " + nom + " = " + date);
    }

    public String toString() {
        String lade = LocalDate.ofEpochDay(
            (long) (Math.ceil((double) date.getTime() / (double) (1000 * 3600 * 24))))
            .format(DateTimeFormatter.ofPattern("dd-MM-yyyy"));
        return nom + " " + " " + prenom + " " + lade;
    }
}

```

► Implémentation du contrôleur REST

```

package com.exemple.hello;

import java.util.concurrent.atomic.AtomicLong;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class BienvenueInviteController {
    @Value("${conference.name: à tout le monde}")
    private String conference;
    private final AtomicLong counter = new AtomicLong();

    @RequestMapping("/") // URL: http://localhost:8080
    public String home() {

```

```

        return "<h1>Bienvenue " + conference + "</h1>";
    }

    @RequestMapping("/bienvenue")
    public Invite bienvenue(@RequestParam(value = "nom", defaultValue = "Palmer")
String nom,
        @RequestParam(value = "prenom", defaultValue = "Jack") String
prenom,
        @RequestParam(value = "email", defaultValue =
"help.dev@spring.io") String email) {
        return new Invite(counter.incrementAndGet(), nom, prenom, email);
    }

    @RequestMapping("/{invite}")
    public String bienvenueInvite(@PathVariable String invite) {
        return "<H1> Bienvenue " + invite + "</H1>";
    }
    @RequestMapping("/test/{nombre}")
    public String testNombre(@PathVariable int nombre) {
        nombre++;
        return "<H1> Nombre reçu + 1: " + nombre + "</H1>";
    }
}

```

L'annotation **@RestController** définit le composant contrôleur REST. Ce composant sera détecté automatiquement.

L'annotation **@RestController** est équivalente aux annotations **@Controller** et **@ResponseBody**.

L'annotation **@ResponseBody** dit au contrôleur que la valeur retournée par chaque méthode constitue le corps de la réponse http retournée au client.

Par défaut, un objet Java retourné sera sérialisé au format JSON.

L'annotation **@RequestMapping("/bienvenue")** définit l'URL relatif de la ressource, ici **http://localhost:8080/bienvenue**.

La méthode annotée **@RequestMapping** est exécutée par défaut pour une commande http **GET**.

L'annotation **@RequestMapping("/{invite}")** définit une variable de nom "invite" dans l'URL. Elle fonctionne en conjonction avec l'annotation **@PathVariable**.

L'annotation **@PathVariable** affecte la variable "invite" de l'URL à la variable Java "invite".

La conversion de type s'effectue automatiquement pour tous les types de base: voir avec **@RequestMapping("/test/{nombre}")** où on récupère une variable de type int.

☛ **Attention: stopper si nécessaire le projet en cours en cliquant sur le bouton rouge avant de relancer l'exécution d'un projet.**

► Exécuter le projet. Tester avec un navigateur les différents URL:

- localhost:8080
- localhost:8080/Dupond
- localhost:8080/bienvenue
- localhost:8080/test/100

► L'accès à l'url **localhost:8080/bienvenue** retourne la chaîne suivante au format json:
`{"id":1,"date":"17-05-2021","nom":"Palmer","prenom":"Jack","email":"help.dev@spring.io"}`

► L'accès à l'url **localhost:8080/bienvenue?nom=Boulon&prenom=Jean** retourne la chaîne suivante au format json:
`{"id":2,"date":"17-05-2021","nom":"Boulon","prenom":"Jean","email":"help.dev@spring.io"}`

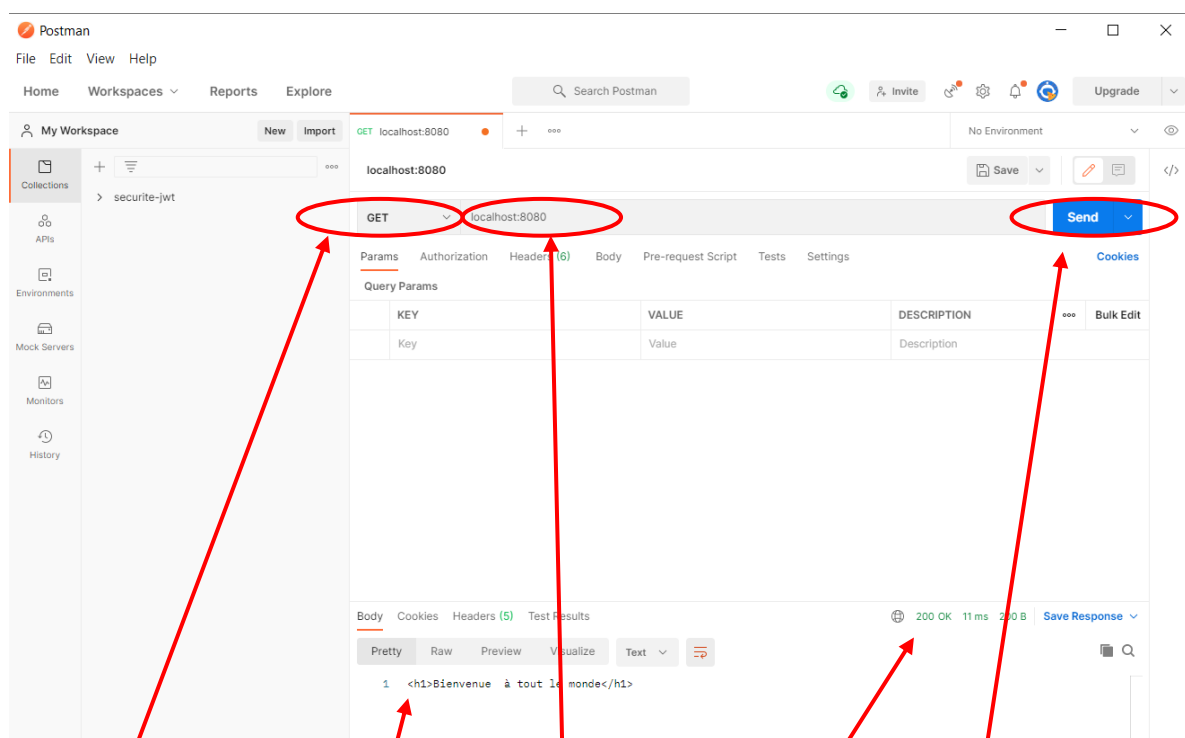
► Effectuez divers tests, justifiez les résultats obtenus.

Utilisation d'un client REST: Postman

De nombreux éditeurs proposent des logiciels client REST sous la forme de plugin de navigateur ou d'applications individuelles.

On peut citer **ARC Advanced Rest Client** de Google Chrome, **RESTClient** pour Firefox...

On présente ici **Postman** installé sous la forme d'une application.



1 Choix de la méthode http.

2 URL de la ressource.

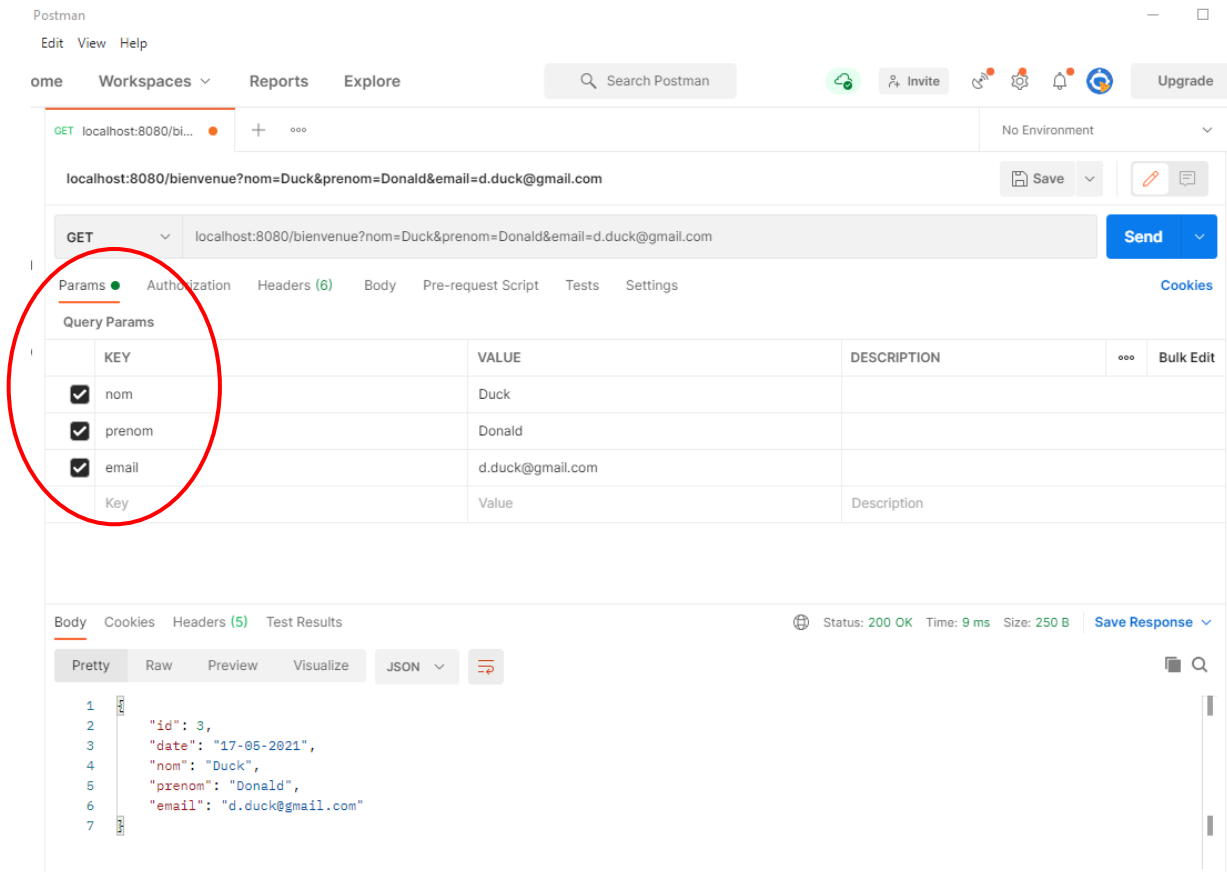
3 Exécution de la méthode.

4 Le "Body" de la réponse

5 Le code de retour

► Il peut être intéressant de supprimer la vue **My Workspace** en cliquant en haut dans la barre de menu sur **View>Toggle Sidebar**.

► Passer des paramètres à la requête GET: il faut utiliser les **Query Params**.



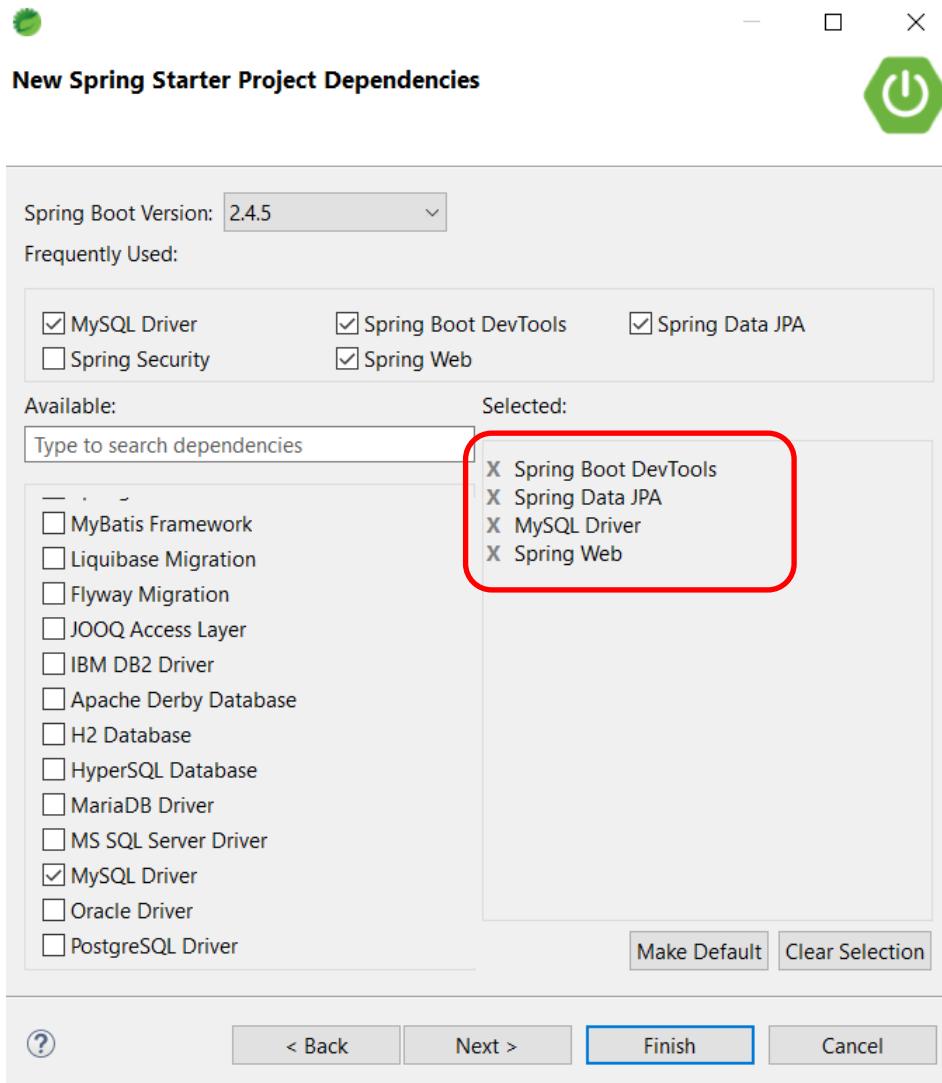
Projet WebService REST avec JPA et MySQL

- Utiliser (ou créer si nécessaire) une base de données **lesinvites** avec la table **invites**.

#	Nom	Type	Interclassement	Attributs	Null	Valeur par défaut	Commentaires	Extra	Action
<input type="checkbox"/> 1	id	int(11)			Non	Aucun(e)		AUTO_INCREMENT	Modifier Supprimer Plus
<input type="checkbox"/> 2	nom	varchar(255)	utf8mb4_general_ci		Non	Aucun(e)			Modifier Supprimer Plus
<input type="checkbox"/> 3	prenom	varchar(255)	utf8mb4_general_ci		Non	Aucun(e)			Modifier Supprimer Plus
<input type="checkbox"/> 4	email	varchar(255)	utf8mb4_general_ci		Non	Aucun(e)			Modifier Supprimer Plus
<input type="checkbox"/> 5	date	date			Non	Aucun(e)			Modifier Supprimer Plus

- Créer un nouveau **Spring Starter Project**, donner lui un nom, cocher **Spring Web**, **Spring Data JPA**, **MySQL Driver** et **Spring Boot DevTools** sur le formulaire des dépendances.

Spring Boot DevTools est très pratique. En effet, avec **Spring Boot DevTools**, le projet est automatiquement rechargé et redémarré dès qu'un changement est fait et sauvegardé dans un des fichiers du CLASSPATH de l'application.



► Remplir le fichier **src/main/resources/application.properties** :

```
spring.jpa.hibernate.ddl-auto=none
spring.datasource.url=jdbc:mysql://localhost:3306/lesinvites?serverTimezone=UTC
spring.datasource.username=admin
spring.datasource.password=admin
```

Remplacer **none** si nécessaire avec la valeur convenable.

► **L'entité Invite : on suppose la clé primaire auto incrémentée**

```
package com.exemple.hello;
import java.io.Serializable;
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.util.Date;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

import com.fasterxml.jackson.annotation.JsonFormat;

@Entity
```



```

@Table(name = "invites")
public class Invite implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    private long id;
    @JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "dd-MM-yyyy")
    private Date date;
    private String nom;
    private String prenom;
    private String email;

    public Invite() {
    }
    public long getId() {
        return this.id;
    }
    public void setId(long id) {
        this.id = id;
    }
    public Date getDate() {
        return this.date;
    }
    public void setDate(Date date) {
        this.date = date;
    }
    public String getNom() {
        return this.nom;
    }
    public void setNom(String nom) {
        this.nom = nom;
    }
    public String getPrenom() {
        return this.prenom;
    }
    public void setPrenom(String prenom) {
        this.prenom = prenom;
    }

    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }

    public Invite(long id,String nom, String prenom, String email) {
        this.id = id;
        this.nom = nom;
        this.prenom = prenom;
        this.email = email ;
        LocalDate localdate = LocalDate.now();
        date = new Date(1000 * 24 * 3600 * localdate.toEpochDay());
        System.out.println("date de " + nom + " = " + date);
    }
    public Invite(String nom, String prenom, String email) {
        super();
        this.nom = nom;
        this.prenom = prenom;
        this.email = email;
    }
}

```

```

        LocalDate localdate = LocalDate.now();
        date = new Date(1000 * 24 * 3600 * localdate.toEpochDay());
    }
    public String toString() {
        String ladate = LocalDate.ofEpochDay((long) (Math.ceil((double)
date.getTime() / (double) (1000 * 3600 * 24))))
            .format(DateTimeFormatter.ofPattern("dd-MM-yyyy"));
        return nom + " " + " " + prenom + " " + ladate;
    }
}

```

► Le repository invité

```

package greta78.cda;

import java.util.ArrayList;
import org.springframework.data.repository.CrudRepository;

public interface InviteRepository extends CrudRepository<Invite, Long> {
    ArrayList<Invite> findByNomAndPrenom(String nom, String prenom);
    ArrayList<Invite> findByNom(String nom);
}

```

► Le contrôleur REST: 1^{ère} version simplifiée

```

package com.exemple.hello;

import java.util.ArrayList;
import org.springframework.beans.factory.annotation.Autowired;
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.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class InvitesController {

    @Autowired // Injection du repository
    private InviteRepository inviteRepository;

    @RequestMapping("/default")
    public Invite getDefault() {
        return new Invite("spring", "sav", "sav.spring@gmail.com");
    }

    @RequestMapping("/all")
    ArrayList<Invite> getAll() {
        return (ArrayList<Invite>) inviteRepository.findAll();
    }

    @GetMapping("/{invite}") // Map requêtes GET
    public ArrayList<Invite> rechercheInviteParGetInvite (
        @PathVariable("invite") String nom) {
        ArrayList<Invite> liste = inviteRepository.findByNom(nom);
        return liste;
    }
}

```

```

    }
    @GetMapping                                // Map requêtes GET
    public ArrayList <Invite> rechercheInviteParGet (@RequestParam String nom
                                                    , @RequestParam String prenom) {
        ArrayList <Invite> liste=inviteRepository.findByNomAndPrenom(nom,prenom);
        return liste;
    }
    @PostMapping(path="/ajoute")                // Map requêtes POST
    public String ajouteInviteParPost (@RequestParam String nom
                                        , @RequestParam String prenom, @RequestParam String email) {
        System.out.println("Dans POST");
        Invite i = new Invite(nom,prenom,email);
        inviteRepository.save(i);
        return "Saved";
    }
}

```

L'annotation **@RequestMapping** accepte un argument pour répondre aux diverses commandes http:

```

@RequestMapping(method = RequestMethod.GET)    ← par défaut
@RequestMapping(method = RequestMethod.POST)
@RequestMapping(method = RequestMethod.DELETE)
@RequestMapping(method = RequestMethod.PUT)

```

Spring Web propose également les annotations suivantes:

- @GetMapping
- @PostMapping
- @PutMapping
- @DeleteMapping

► Exécutez le projet.

► Testez les différents URL, chaque URL fournissant un web service est appelé un **end-point**. Vérifiez les résultats affichés dans Postman et notamment le code de retour.

GET localhost:8080/default retourne:

```
{ "id":0,"date":"17-05-2021","nom":"Palmer","prenom":"Jack","email":"sav.spring@gmail.com" }
```

GET localhost:8080/all retourne la liste des invités au format json:

```
[{ "id":1,"date":"17-03-2020","nom":"Dupond","prenom":"Jean","email":"jdupond@gmail.com" },
{ "id":3,"date":"16-03-2020","nom":"Lameche","prenom":"Miloud","email":"mlameche@orange.fr"},..... ]
```

GET localhost:8080/lameche retourne:

```
[{ "id":3,"date":"16-03-2020","nom":"Lameche","prenom":"Miloud","email":"mlameche@orange.fr" }]
```

GET localhost:8080/lameches retourne un body vide avec le code 200 renvoyé.

Conclure.

► Ajout d'un invité avec la commande **POST**.

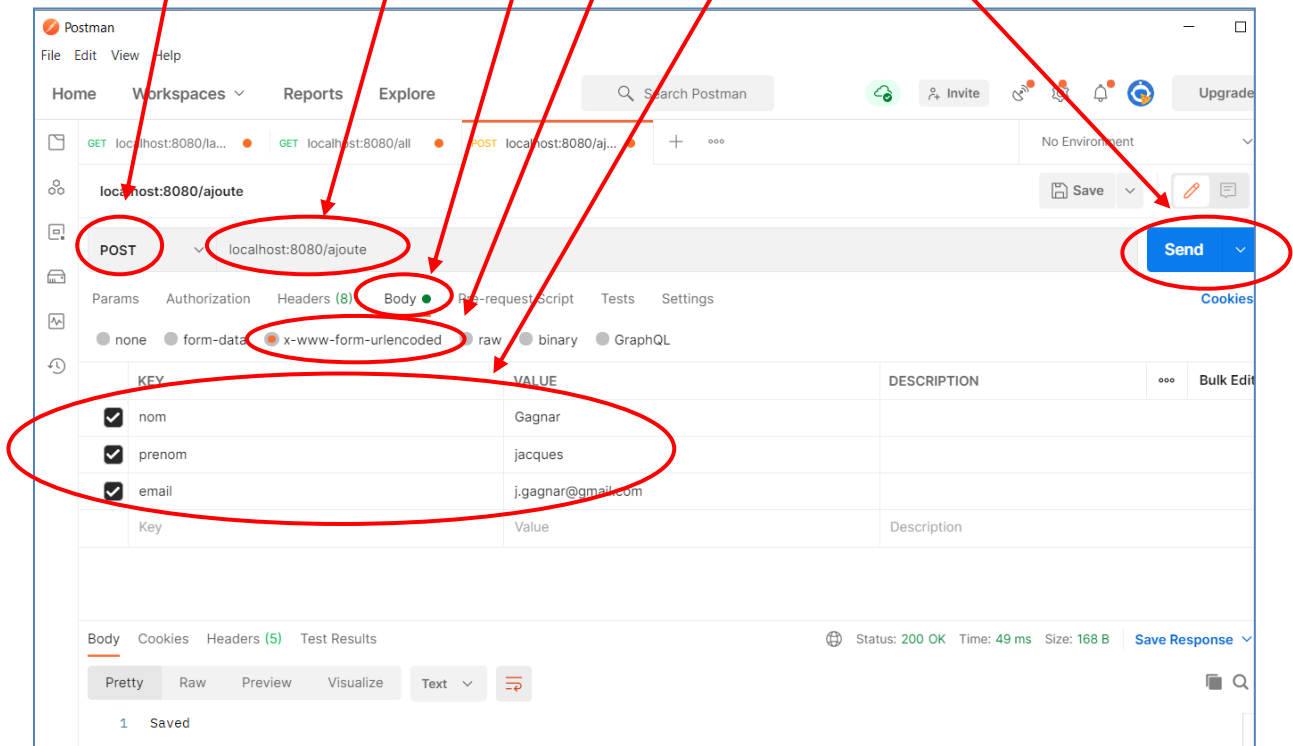
1 Sélectionnez **POST** **2** Tapez l' URL localhost:8080/ajoute

3 Sélectionnez **Body**

4 Sélectionnez **x-www-form-urlencoded**

5 Entrer les paramètres et leurs valeurs

6 Exécutez la requête



Après l'exécution, le client reçoit bien "Saved"; le code retourné est 200; vérifiez l'ajout de l'invité dans la table.

Les codes de retour ne conviennent pas pour plusieurs de ces end-point.

Il faut modifier le contrôleur REST pour respecter les codes de retour normalisés.

Un contrôleur REST 2^{ème} version

Ce contrôleur assure les opérations CRUD en utilisant le repository précédent, il respecte les codes de retour standard habituels.

La documentation Microsoft suivante sur les méthodes HTTP détaille les codes retournés au client pour chaque méthode.

Méthodes GET

En général, une méthode GET réussie renvoie le code d'état HTTP 200 (OK). Si la ressource est introuvable, la méthode doit renvoyer 404 (Introuvable).

Méthodes POST

Si une méthode POST crée une ressource, elle renvoie le code d'état HTTP 201 (Créé). L'URI de la nouvelle ressource est inclus dans l'en-tête **Location** de la réponse. Le corps de la réponse contient une représentation de la ressource.

Si la méthode effectue des opérations de traitement, mais ne crée pas de ressource, elle peut renvoyer le code d'état HTTP 200 et inclure le résultat de l'opération dans le corps de la réponse. Ou bien, en l'absence de résultat à renvoyer, la méthode peut renvoyer le code d'état HTTP 204 (Pas de contenu) sans corps de réponse.

Si le client place des données non valides dans la requête, le serveur doit renvoyer le code d'état HTTP 400 (Demande incorrecte). Le corps de la réponse peut contenir des informations supplémentaires sur l'erreur ou un lien vers un URI qui fournit plus de détails.

Méthodes PUT

Si une méthode PUT crée une ressource, elle renvoie le code d'état HTTP 201 (Créé), comme pour la méthode POST. Si la méthode met à jour une ressource existante, elle renvoie 200 (OK) ou 204 (Pas de contenu). Dans certains cas, mettre à jour une ressource existante peut se révéler impossible. Dans ce cas, envisagez de renvoyer le code d'état HTTP 409 (Conflit).

Méthodes DELETE

Si l'opération de suppression est réussie, le serveur web doit répondre avec un code d'état HTTP 204 indiquant que le processus a été géré correctement mais que le corps de la réponse ne contient aucune information supplémentaire. Si la ressource n'existe pas, le serveur web peut renvoyer HTTP 404 (Introuvable).

► Créez un nouveau **Spring Starter Project**, donnez lui un nom, cochez **Spring Web**, **Spring Data JPA**, **MySQL Driver** et **Spring Boot DevTools** sur le formulaire des dépendances.

► Copiez dans ce projet les fichiers **Invite.java**, **InviteRepository.java** et **application.properties** du projet précédent.

La classe ResponseEntity<T>

Spring Web propose la classe **ResponseEntity<T>** pour créer la réponse renvoyée au client.

Cette classe permet de préciser tous les éléments de la réponse HTTP renvoyée au client: le code de retour (status), l'entête et le Body.

Etude d'un exemple:

URL : **GET localhost:8080/lameche**
ou **GET localhost:8080/lameche1**

```
@GetMapping("/{invite}") // Map requêtes GET
public ResponseEntity<Invite> rechercheInviteParGetInvite (
    @PathVariable("invite") String nom) {
    ArrayList<Invite> liste = inviteRepository.findByNom(nom);
    if (liste.size()==0)
        return ResponseEntity.notFound().build();
    else
        return ResponseEntity.ok(liste.get(0));
}
```

☞ On suppose qu'il n'y a pas de doublons dans la table.

inviteRepository.findByNom(nom); retourne une liste vide ou une liste avec l'invité recherché.

Si l'invité est trouvé, **ResponseEntity.ok(liste.get(0))**; retourne une réponse avec le code de retour 200 et l'invité trouvé au format json dans le body.

Si aucun invité n'est trouvé, **ResponseEntity.notFound()** retourne un builder avec le code 404 et **ResponseEntity.notFound().build()** construit la réponse avec le code 404 et le body vide.

☞ **Pour la methode POST, avant d'insérer un nouvel invite il faut s'assurer de ne pas créer de doublons dans la table.**

On suppose qu'un doublon est défini par 2 invités qui auraient le même nom, le même prénom et le même email. Il faut donc faire cette recherche avant d'insérer le nouvel invite.

On ajoute pour cela dans **InviteRepository** la méthode suivante **findByNomAndPrenomAndEmail(...)** écrite en gras ci-dessous.

```
package greta78.cda;

import java.util.ArrayList;
import org.springframework.data.repository.CrudRepository;

public interface InviteRepository extends CrudRepository<Invite, Long> {
    ArrayList<Invite> findByNomAndPrenom(String nom, String prenom);
    ArrayList<Invite> findByNom(String nom);
    Invite findByNomAndPrenomAndEmail(String nom, String prenom, String email);
}
```

► Le contrôleur

On modifie le contrôleur en appliquant les règles habituelles des web services.

La méthode POST reçoit les attributs de l'objet Invite à créer, en cas de réussite, elle retourne l'indication HTTP de la création de l'objet (code de retour 201) ainsi que l'URI du nouvel objet.

```
package com.exemple.hello;

import java.net.URI;
import java.util.ArrayList;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
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.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.servlet.support.ServletUriComponentsBuilder;

@RestController
public class InvitesController {

    @Autowired // Injection du repository
    private InviteRepository inviteRepository;

    @RequestMapping("/defaut")
    public Invite getDefault() {
        return new Invite("Palmer", "Jack", "sav.spring@gmail.com");
    }

    @RequestMapping("/all")
```

```

    ArrayList<Invite> getAll() {
        return (ArrayList<Invite>) inviteRepository.findAll();
    }
    @GetMapping("/{invite}") // Map requêtes GET
    public ResponseEntity<Invite> rechercheInviteParGetInvite (
        @PathVariable("invite") String nom) {
        ArrayList<Invite> liste = inviteRepository.findByNom(nom);
        if (liste.size()==0)
            return ResponseEntity.notFound().build();
        else
            return ResponseEntity.ok(liste.get(0));
    }
    @GetMapping // Map requêtes GET
    public ResponseEntity<Invite> rechercheInviteParGet (@RequestParam String nom
        , @RequestParam String prenom) {
        ArrayList<Invite>
liste=inviteRepository.findByNomAndPrenom(nom,prenom);
        if (liste.size()==0)
            return ResponseEntity.notFound().build();
        else
            return ResponseEntity.ok(liste.get(0));
    }
    @PostMapping(path="/ajoute") // Map requêtes POST
    public ResponseEntity<Void> ajouteInvite (@RequestParam String nom
        , @RequestParam String prenom, @RequestParam String email) {
        Invite r =
inviteRepository.findByNomAndPrenomAndEmail(nom,prenom,email);
        if (r != null)
            return ResponseEntity.noContent().build();
        Invite i = new Invite(nom,prenom,email);
        i = inviteRepository.save(i);
        URI location = ServletUriComponentsBuilder
            .fromCurrentRequest()
            .path("/{id}")
            .buildAndExpand(i.getId())
            .toUri();
        return ResponseEntity.created(location).build();
    }
    @DeleteMapping // Map requêtes DELETE
    public ResponseEntity<Void> supprimerInvite(@RequestParam String nom
        , @RequestParam String prenom) {
        ArrayList<Invite>
liste=inviteRepository.findByNomAndPrenom(nom,prenom);
        if (liste.size()==0)
            return ResponseEntity.notFound().build();
        else {
            inviteRepository.delete(liste.get(0));
            return
ResponseEntity.status(HttpStatus.NO_CONTENT).build();
        }
    }
    @PutMapping(path="/modifie") // Map requêtes PUT
    public ResponseEntity<Void> modifieEmailInvite (@RequestParam String nom
        , @RequestParam String prenom, @RequestParam String email) {
        ArrayList<Invite> liste =
inviteRepository.findByNomAndPrenom(nom,prenom);
        if (liste.size() != 0) {
            Invite i = liste.get(0);
            i.setEmail(email);

```

```

        inviteRepository.save(i);
        return ResponseEntity.ok().build();
    }
    Invite i = new Invite(nom, prenom, email);
    i = inviteRepository.save(i);
    URI location = ServletUriComponentsBuilder
        .fromCurrentContextPath()
        .path("/{id}")
        .buildAndExpand(i.getId())
        .toUri();
    return ResponseEntity.created(location).build();
}

```

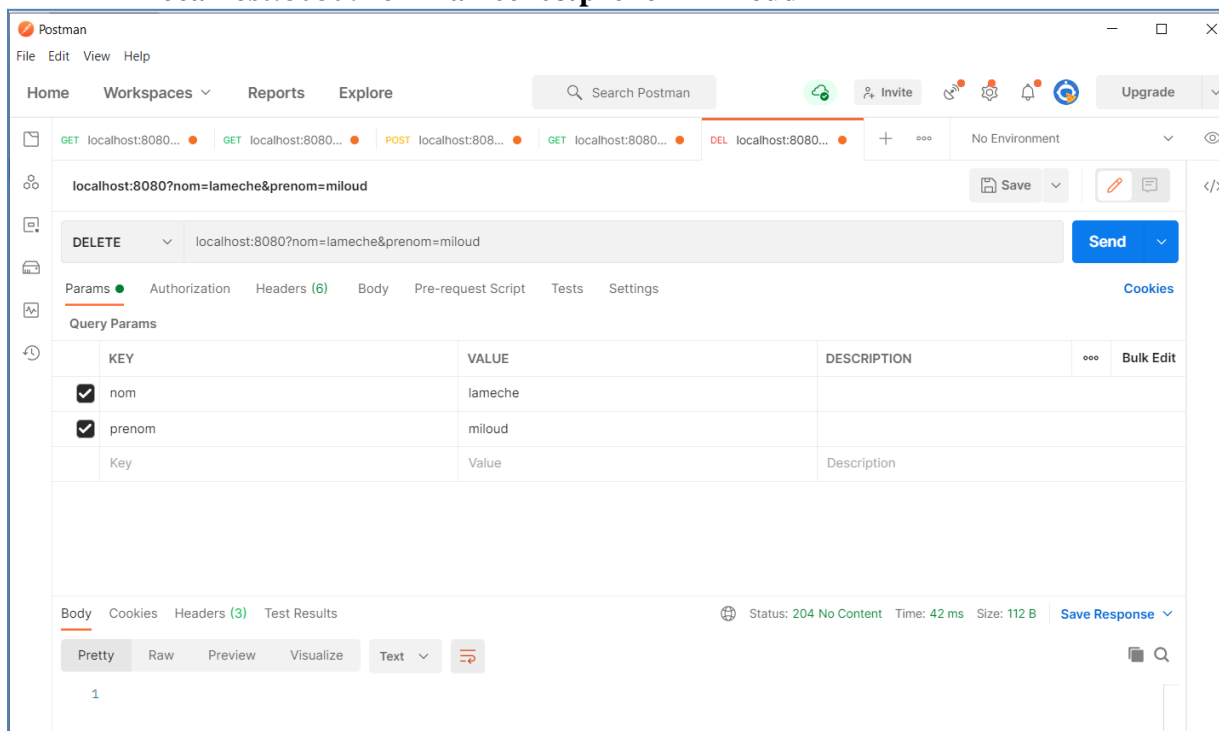
Le type **ResponseEntity<Void>** signifie que la réponse à un "body" vide.

► Tester les différents URL possible.

Quelques exemples de résultats sont présentés ci-dessous.

► Exécution de la méthode DELETE:

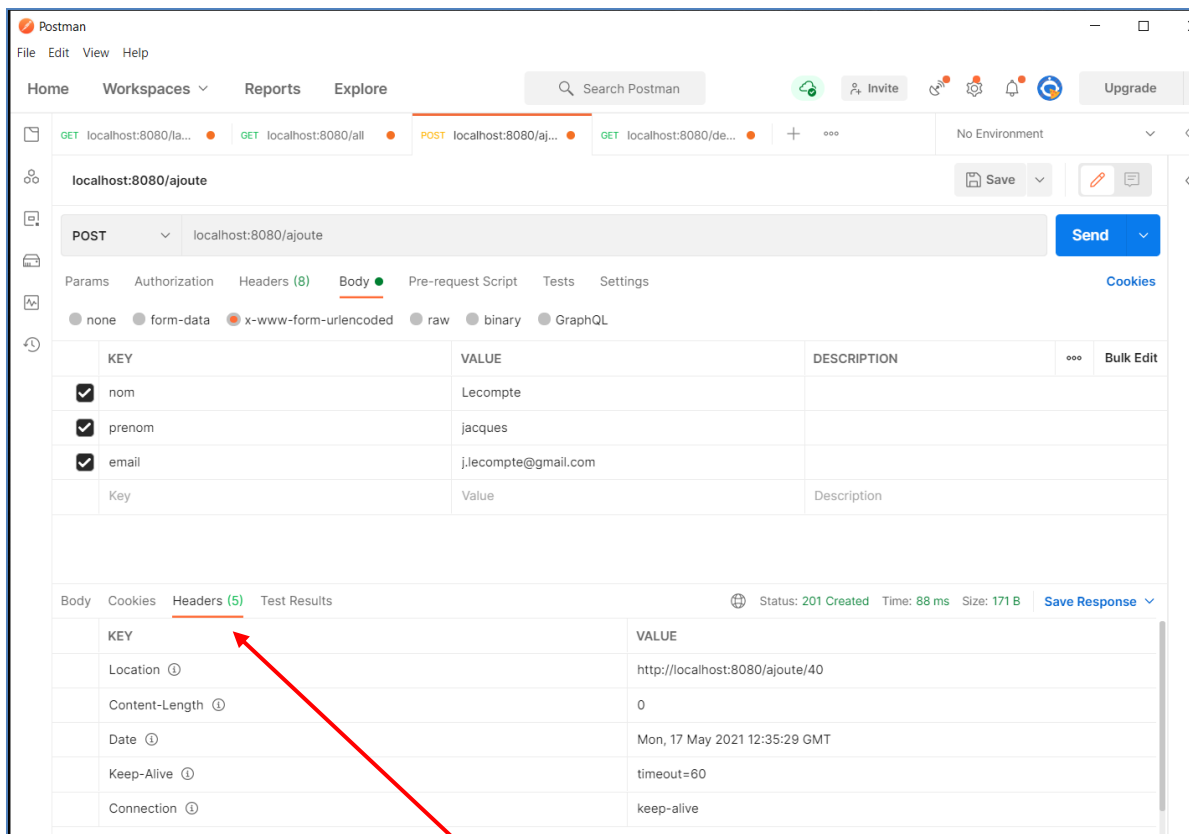
DELETE localhost:8080?nom=lameche&prenom=miloud



Le status retourné est bien 204.

► Vérifiez maintenant qu'une exécution de cette même requête retourne 404 Not Found.

► Exécution de la méthode POST:



Le **Body** est vide, cliquez sur **Headers** pour voir le champ **Location**.

► Ajout d'une méthode POST recevant un objet Invite au format Json

La méthode POST reçoit un objet json; en cas de réussite, elle retourne l'indication http de la création de l'objet (code de retour 201) ainsi que l'URI du nouvel objet.

```
// ajoute un invite, l'objet est reçu au format json
@PostMapping(value = "/ajoutejson")
public ResponseEntity<Void> ajouterInvite(@RequestBody Invite invite) {
    Invite r = inviteRepository.findByNomAndPrenomAndEmail(invite.getNom(),
    invite.getPrenom(), invite.getEmail());
    if (r != null)
        return ResponseEntity.noContent().build();
    invite.setDate(Date.from(Instant.now()));
    Invite inviteAjoute = inviteRepository.save(invite);

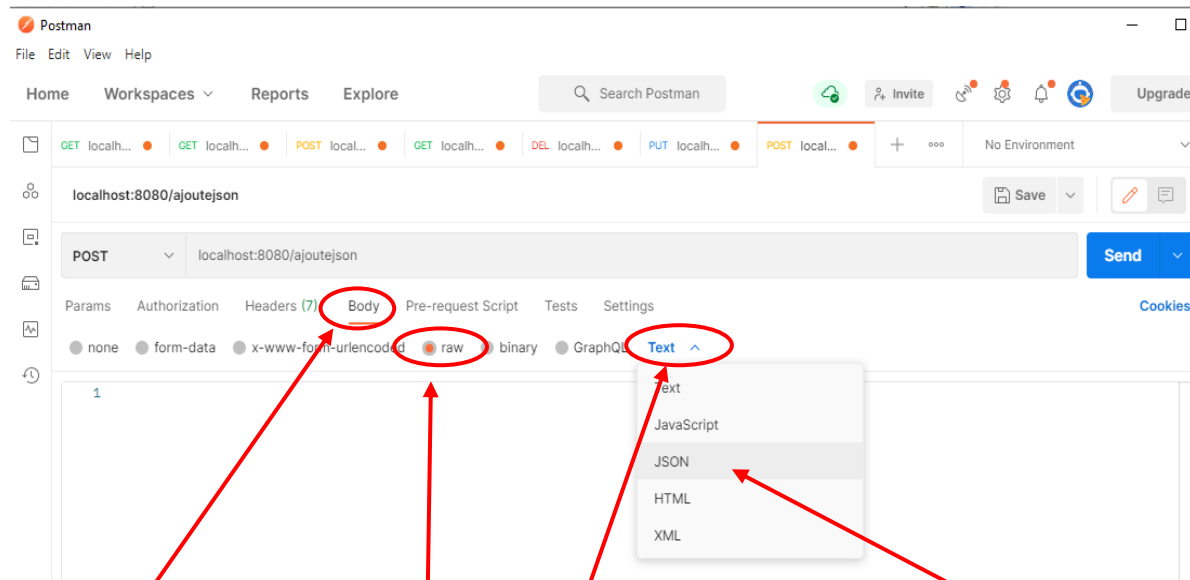
    URI location = ServletUriComponentsBuilder
        .fromCurrentRequest().path("/{id}")
        .buildAndExpand(inviteAjoute.getId()).toUri();
    return ResponseEntity.created(location).build();
}
```

La classe **ResponseEntity** hérite de **HttpEntity** qui permet de définir le code HTTP à retourner. On retourne le code 204 si le produit n'est pas créé.

Si le produit est créé, on invoque la méthode **created()** de **ResponseEntity** en lui passant l'URI de la ressource créée. Pour cela:

- on récupère un objet **ServletUriComponentsBuilder** par **ServletUriComponentsBuilder.fromCurrentRequest()** correspondant à la requête,

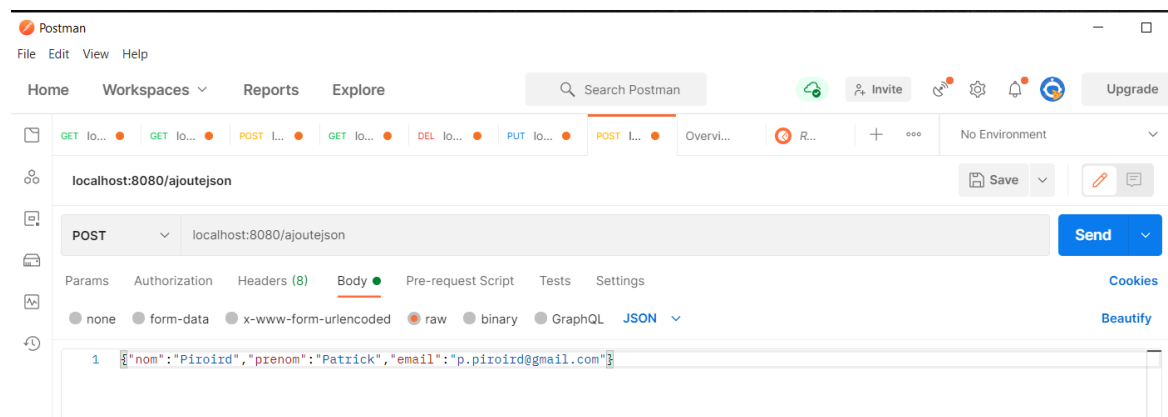
- la méthode **path("/{id}")**, ajoute le chemin relatif à l'URI du composant précédent et retourne un **UriComponentBuilder**,
- la méthode **buildAndExpand(inviteAjoute.getId())**, affecte l'id de l'invité créé à la partie variable précédente et retourne un **UriComponents**,
- la méthode **toUri()** de l'**UriComponents** précédent retourne l'URI ainsi créé.



Sélectionnez **Body** Sélectionnez **raw** Ouvrez **Text** puis sélectionnez **JSON**

Puis tapez les paramètres du nouvel invité au format JSON et exécutez.

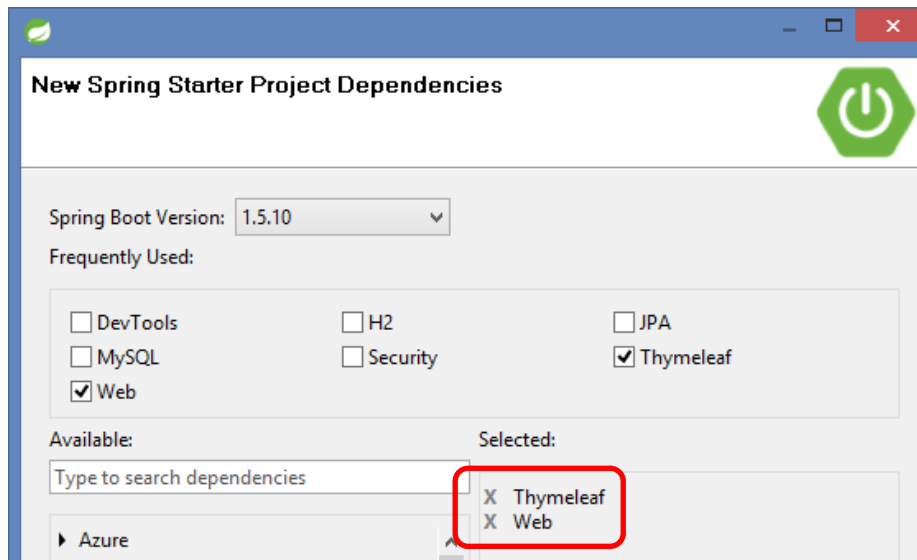
```
{ "nom": "Piroird", "prenom": "Patrick", "email": "p.piroird@gmail.com" }
```



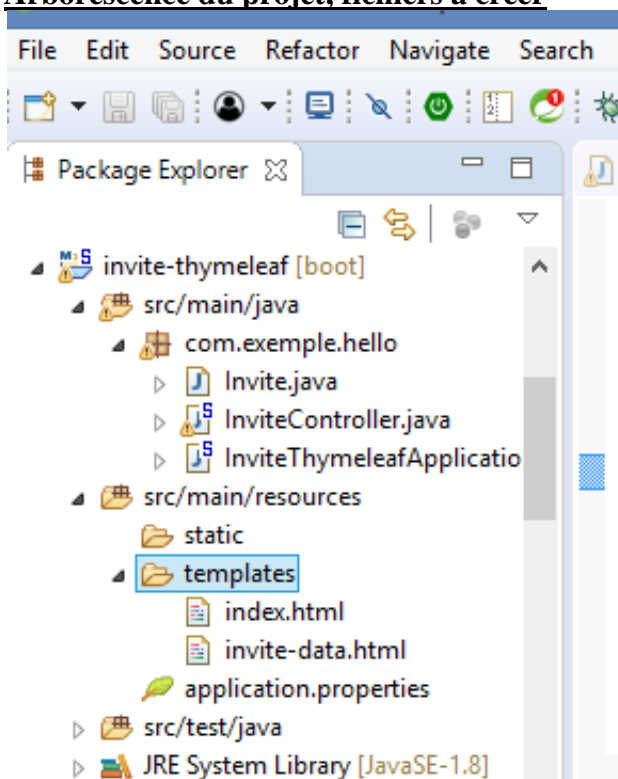
Voir le lien suivant pour les codes de retour des services REST

<https://restfulapi.net/http-status-codes/>

Projet WebService REST avec Thymeleaf



Arborescence du projet, fichiers à créer



► Créer le fichier templates/index.html

```
<html>
<head>
<title>Page d'index</title>
</head>
<body>
    <form action="save" method="post">
```

```

        <table>
            <tr>
                <td><label>Nom :</label></td>
                <td><input type="text" name="nom"></input></td>
            </tr>
            <tr>
                <td><label>Prénom :</label></td>
                <td><input type="text" name="prenom"></input></td>
            </tr>
            <tr>
                <td><label>Email :</label></td>
                <td><input type="text" name="email"></input></td>
            </tr>
            <tr>
                <td></td>
                <td><input type="submit" value="Validez"></input></td>
            </tr>
        </table>
    </form>
</body>
</html>

```

► Créer le fichier templates/invite-data.html

```

<html xmlns:th="http://thymeleaf.org">
<table>
    <tr>
        <td><h4>Nom :</h4></td>
        <td><h4 th:text="${invite.nom}"></h4></td>
    </tr>
    <tr>
        <td><h4>Prénom :</h4></td>
        <td><h4 th:text="${invite.prenom}"></h4></td>
    </tr>
    <tr>
        <td><h4>Email :</h4></td>
        <td><h4 th:text="${invite.email}"></h4></td>
    </tr>
</table>
</html>

```

► Créer le modèle Invite.java

```

package com.exemple.hello;

import java.io.Serializable;
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.util.Date;

import com.fasterxml.jackson.annotation.JsonFormat;

```

```

public class Invite implements Serializable {
    private static final long serialVersionUID = 1L;

    private long id;
    @JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "dd-MM-yyyy")
    private Date date;
    private String nom;
    private String prenom;
    private String email;

    public Invite() {
    }

    public long getId() {
        return this.id;
    }

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

    public Date getDate() {
        return this.date;
    }

    public void setDate(Date date) {
        this.date = date;
    }

    public String getNom() {
        return this.nom;
    }

    public void setNom(String nom) {
        this.nom = nom;
    }

    public String getPrenom() {
        return this.prenom;
    }

    public void setPrenom(String prenom) {
        this.prenom = prenom;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}

```

```

    public Invite(long id,String nom, String prenom, String email) {
        this.id = id;
        this.nom = nom;
        this.prenom = prenom;
        this.email = email ;
        LocalDate localdate = LocalDate.now();
        date = new Date(1000 * 24 * 3600 * localdate.toEpochDay());
        System.out.println("date de " + nom + " = " + date);
    }

    public String toString() {
        String ldate = LocalDate.ofEpochDay((long) (Math.ceil((double) date.getTime()
/ (double) (1000 * 3600 * 24))))
        .format(DateTimeFormatter.ofPattern("dd-MM-yyyy"));
        return nom + " " + " " + prenom + " " + ldate;
    }
}

```

► Créer le contrôleur **InviteController.java**

```

package com.exemple.hello;

```

```

import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.stereotype.Controller;

```

```

@Controller

```

```

public class InviteController {
    @RequestMapping("/")
    public String index() {
        return "index";
    }

    @RequestMapping(value = "/save", method = RequestMethod.POST)
    public ModelAndView save(@ModelAttribute Invite invite) {
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.setViewName("invite-data");
        modelAndView.addObject("invite", invite);
        return modelAndView;
    }
}

```

► Exécuter l'application.

► L'affichage de l'URL **localhost:8080** dans le navigateur donne le formulaire suivant:

Page d'index

localhost:8080

Nom :

Prénom :

Email :

Validez

Page d'index

localhost:8080

Nom :

Prénom :

Email :

Validez

Remplir les champs puis valider.

Cela donne :

localhost:8080/save

localhost:8080/save

Nom : **Durand**

Prénom : **Jean**

Email : **j.durand@gmail.com**

Améliorer l'application en gardant le même formulaire de saisie et en ajoutant dans la vue récapitulative la date de la création de l'invité.

localhost:8080/save

localhost:8080/save

Nom : **Durand**

Prénom : **Jean**

Email : **j.durand@gmail.com**

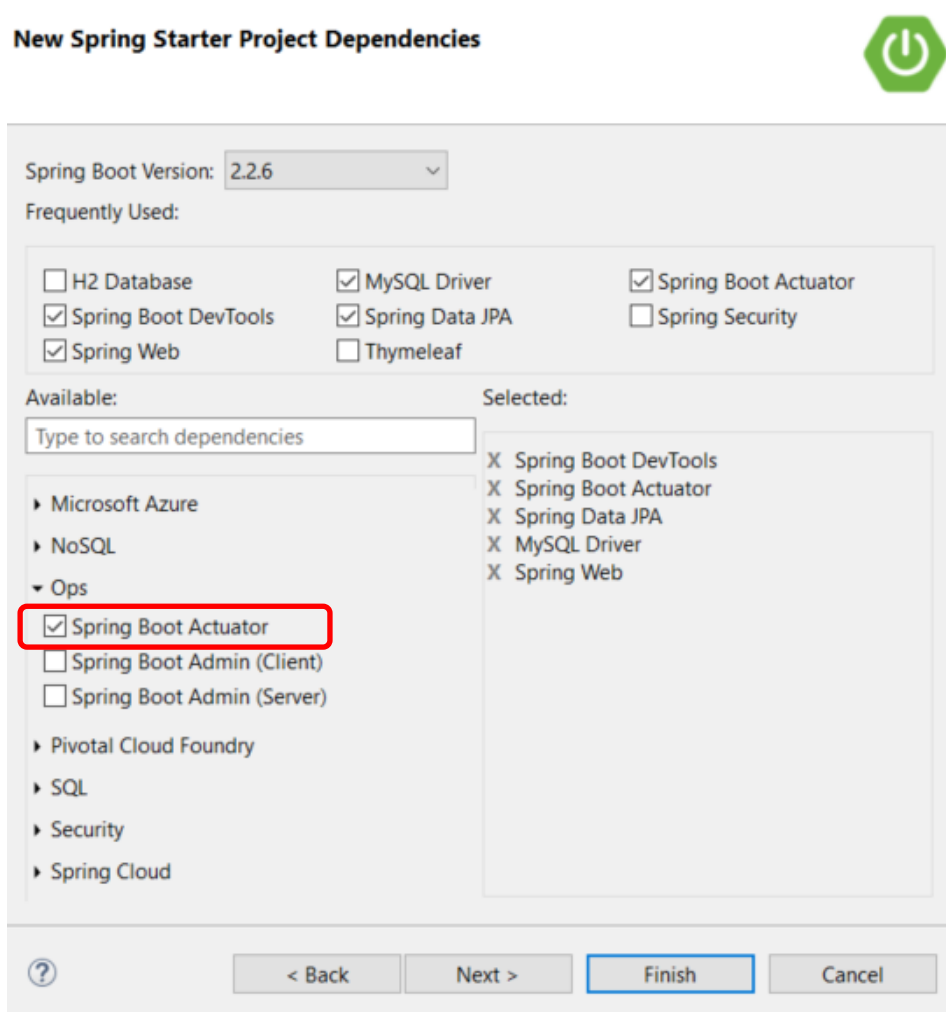
Date : **20-mars-2018**

Superviser un projet WebService REST-JPA-MySQL

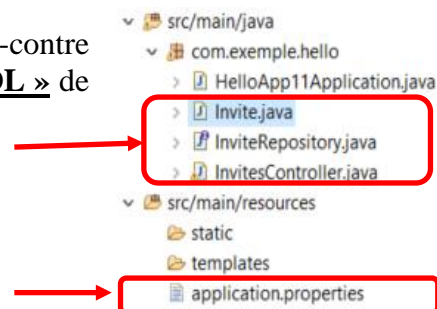
Spring Boot comprend un certain nombre de fonctionnalités supplémentaires pour aider à surveiller et à gérer une application lorsqu'elle est mise en production. On peut choisir de gérer et de surveiller l'application à l'aide de points de terminaison HTTP ou avec JMX. L'audit, l'intégrité et la collecte de métriques peuvent également être appliqués automatiquement à l'application.

Pour cela, il faut intégrer le module **spring-boot-actuator**, on développe un exemple en créant un nouveau projet Spring Boot qui intègre cette fonctionnalité.

Cela entraînera automatiquement la création de services rest spécifiques et accessibles à des URI bien précis appelés points de terminaison (endpoint).

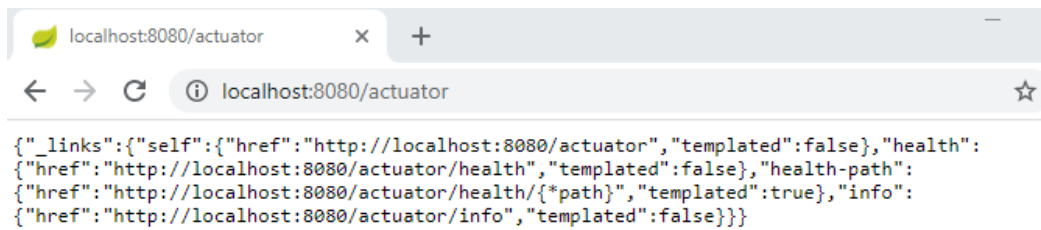


► Copiez dans ce nouveau projet les fichiers indiqués ci-contre du projet « **Projet Webservice REST avec JPA, MySQL** » de la page 16.



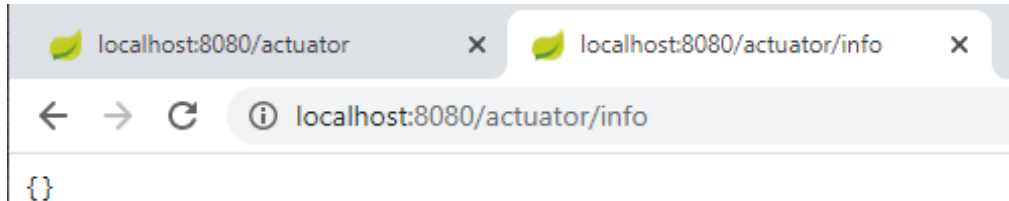
► Vérifiez le fonctionnement de l'application avec l'URI **localhost:8080/all**

► Ouvrir la page **http://localhost:8080/actuator**



Cette page affiche les services offerts par défaut et leurs uri correspondants: le service **health** et le service **info** sont exposés (accessibles) **par défaut**.

► Afficher le service **info** :



Aucune information n'est affichée par défaut.

Ajout d'informations sur l'application, le endpoint /info

► On ajoute es informations encadrées en rouge dans le fichier **application.properties**.

```

spring.jpa.hibernate.ddl-auto=none
spring.datasource.url=jdbc:mysql://localhost:3306/lesinvites?serverTimezone=Europe/Paris
spring.datasource.username=admin
spring.datasource.password=admin

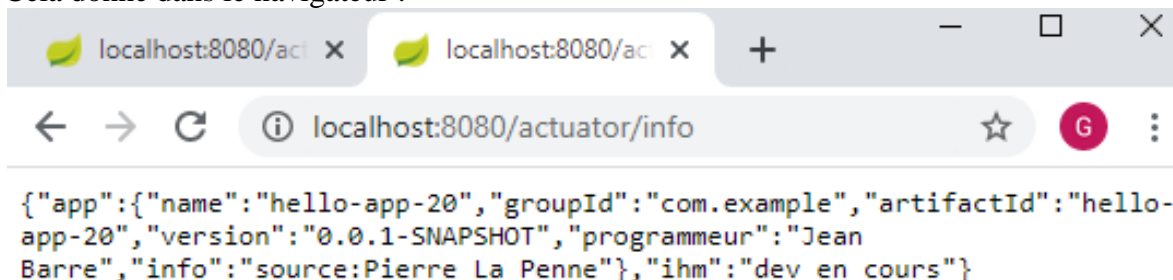
```

```

info.app.name = @project.name@
info.app.groupId = @project.groupId@
info.app.artifactId = @project.artifactId@
info.app.version = @project.version@
info.app.programmeur=Jean Barre
info.app.info=source:Pierre La Penne
info.ihm=dev en cours

```

Cela donne dans le navigateur :

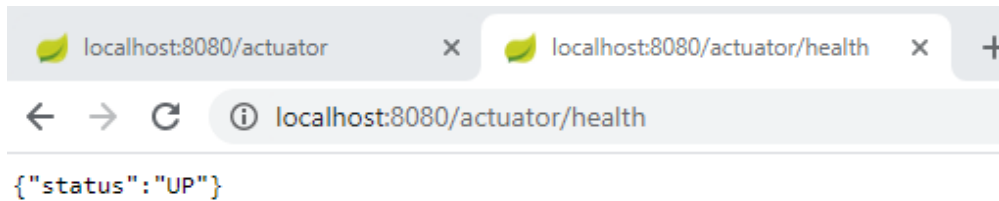


Ces informations sont publiées au format JSON.

On peut utiliser des noms de variables prédéfinies @xxxx@ ou des variables « personnelles ».

Configuration de base du service health, le endpoint /health

► Afficher le service **health** :



Il nous indique que l'application est **UP**, c.a.d en activité.

► Compléter les informations affichées: ajouter la ligne suivante dans le fichier **application.properties**:
`management.endpoint.health.show-details=always`

► Exécuter le end-point **localhost:8080/actuator/health**

```
{
  "status": "UP",
  "components": {
    "db": {
      "status": "UP",
      "details": {
        "database": "MySQL",
        "validationQuery": "isValid()"
      }
    },
    "diskSpace": {
      "status": "UP",
      "details": {
        "total": 1000203087872,
        "free": 869784797184,
        "threshold": 10485760,
        "exists": true
      }
    },
    "ping": {
      "status": "UP"
    }
  }
}
```

Configuration d'un service health, le endpoint /health

On se propose de simuler un dysfonctionnement de l'application dans le cas où la table invité contiendrait des doublons :

- **ON** s'il n'y a pas d'invités identiques dans la table invité,
- **OFF** si on trouve des doublons dans la table invité.

La recherche des **doublons** porte sur les champs **nom, prénom et email**. Les doublons recherchés sont relatifs car on peut supposer qu'ils ont une clef différente (clef auto incrémentée) mais aussi une date de création différente.

La requête SQL :

SELECT * FROM `invites` group by nom,prenom,email having count(*)>1

► La requête JPQL à ajouter dans le fichier `InviteRepository.java` :

```
@Query("select i from Invite i group by i.nom,i.prenom,i.email having count(*)>1")
Collection<Invite> rechercherDoublons() ;
```

Chaque service "health" doit implémenter l'interface **HealthIndicator** et définir la méthode **health()**.

► Ajouter la classe suivante dans le projet :

```
package com.exemple.hello;

import java.util.Collection;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.actuate.health.Health;
import org.springframework.boot.actuate.health.HealthIndicator;
import org.springframework.stereotype.Component;

@Component
public class HealthCheckBDD implements HealthIndicator {

    private final String message_key = "Base de données:";
    @Autowired // Injection du repository
    private InviteRepository inviteRepository;

    @Override
    public Health health() {
        int errorCode = check(); // perform some specific health check
        if (errorCode != 0) {
            return Health.down().withDetail(message_key, "Doublons détectés").build();
        }
        return Health.up().withDetail(message_key, "Pas de doublons détectés").build();
    }

    public int check() {
        // Our logic to check health
        Collection<Invite> doublons = inviteRepository.rechercherDoublons();
        if (doublons.size() == 0)
            return 0;
        else return 1;
    }
}
```

La méthode **check()** retourne 0 si pas de doublons ou 1 si des doublons sont présents.

La méthode **Health health()** doit être définie dans la classe, elle retourne **Health.down().build()** si le système dysfonctionne (doublons présents) ou **Health.up().build()** dans le cas contraire.

► Tester **<http://localhost:8080/actuator/health>** sans doublon puis avec en ajoutant des doublons dans la table invité.

Résultats observables avec **doublon** dans la base: le status global est down.

```
{
  "status": "DOWN",
  "components": {
    "db": {
```

```

        "status": "UP",
        "details": {
            "database": "MySQL",
            "validationQuery": "isValid()"
        }
    },
    "diskSpace": {
        "status": "UP",
        "details": {
            "total": 1000203087872,
            "free": 869784780800,
            "threshold": 10485760,
            "exists": true
        }
    },
    "healthCheckBDD": {
        "status": "DOWN",
        "details": {
            "Base de données": "Doublons détectés"
        }
    },
    "ping": {
        "status": "UP"
    }
}
}

```

► Ajouter la classe suivante dans le projet pour obtenir la santé d'un web service.

```

package com.exemple.hello;

import org.springframework.boot.actuate.health.Health;
import org.springframework.boot.actuate.health.HealthIndicator;
import org.springframework.stereotype.Component;
import org.springframework.web.client.RestTemplate;

@Component
public class HealthCheckGetOneIndividu implements HealthIndicator{

    @Override
    public Health health() {
        RestTemplate restTemplate = new RestTemplate();
        String message_key = "URL par défaut:";
        Invite uninvite =
restTemplate.getForObject("http://localhost:8080/default", Invite.class);
        if (uninvite.getNom().equals("Palmer") &&
uninvite.getPrenom().equals("Jack")) {
            return Health.up().withDetail(message_key, "fonctionne
bien").build();
        }
        else
            return Health.up().withDetail(message_key, "erreur").build();
        }
    }
}

```

► Tester **http://localhost:8080/actuator/health** sans erreur sur l'invité retourné par défaut puis en créant une erreur.

Résultats observables sans erreur sur l'end-point /defaut et avec **doublon** dans la base: le status global est down.

```
"status": "DOWN",
"components": {
  "db": {
    "status": "UP",
    "details": {
      "database": "MySQL",
      "validationQuery": "isValid()"
    }
  },
  "diskSpace": {
    "status": "UP",
    "details": {
      "total": 1000203087872,
      "free": 869784678400,
      "threshold": 10485760,
      "exists": true
    }
  },
  "healthCheckBDD": {
    "status": "DOWN",
    "details": {
      "Base de données": "Doublons détectés"
    }
  },
  "healthCheckGetOneIndividu": {
    "status": "UP",
    "details": {
      "URL par défaut": "fonctionne bien"
    }
  },
  "ping": {
    "status": "UP"
  }
}
```

Liste des endpoints possibles

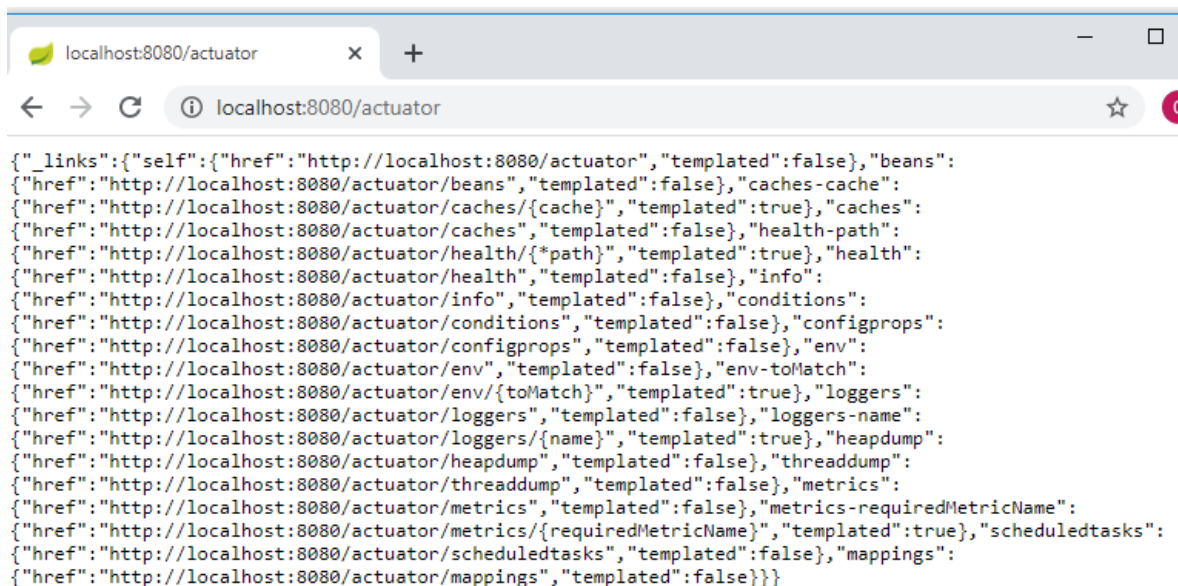
ID	Description
auditevents	Exposes audit events information for the current application. Requires an <code>AuditEventRepository</code> bean.
beans	Displays a complete list of all the Spring beans in your application.
caches	Exposes available caches.
conditions	Shows the conditions that were evaluated on configuration and auto-configuration classes and the reasons why they did or did not match.
configprops	Displays a collated list of all <code>@ConfigurationProperties</code> .
env	Exposes properties from Spring's <code>ConfigurableEnvironment</code> .
flyway	Shows any Flyway database migrations that have been applied. Requires one or more <code>Flyway</code> beans.
health	Shows application health information.
httptrace	Displays HTTP trace information (by default, the last 100 HTTP request-response exchanges). Requires an <code>HttpTraceRepository</code> bean.
info	Displays arbitrary application info.
integrationgraph	Shows the Spring Integration graph. Requires a dependency on <code>spring-integration-core</code> .
loggers	Shows and modifies the configuration of loggers in the application.
liquibase	Shows any Liquibase database migrations that have been applied. Requires one or more <code>Liquibase</code> beans.
metrics	Shows 'metrics' information for the current application.
mappings	Displays a collated list of all <code>@RequestMapping</code> paths.
scheduledtasks	Displays the scheduled tasks in your application.
sessions	Allows retrieval and deletion of user sessions from a Spring Session-backed session store. Requires a Servlet-based web application using Spring Session.
shutdown	Lets the application be gracefully shutdown. Disabled by default.
threaddump	Performs a thread dump.

Par défaut, tous les endpoint sont valides mais seuls les endpoint **health** et **info** sont **exposés**.

► Il faut ajouter l'instruction suivante dans le fichier **application.properties** si on veut **exposer tous les endpoint** :

management.endpoints.web.exposure.include=*

► Ouvrir la page **http://localhost:8080/actuator**



```
{
  "_links": {
    "self": {
      "href": "http://localhost:8080/actuator",
      "templated": false
    },
    "beans": {
      "href": "http://localhost:8080/actuator/beans",
      "templated": false
    },
    "caches-cache": {
      "href": "http://localhost:8080/actuator/caches/{cache}",
      "templated": true
    },
    "caches": {
      "href": "http://localhost:8080/actuator/caches",
      "templated": false
    },
    "health-path": {
      "href": "http://localhost:8080/actuator/health/{path}",
      "templated": true
    },
    "health": {
      "href": "http://localhost:8080/actuator/health",
      "templated": false
    },
    "info": {
      "href": "http://localhost:8080/actuator/info",
      "templated": false
    },
    "conditions": {
      "href": "http://localhost:8080/actuator/conditions",
      "templated": false
    },
    "configprops": {
      "href": "http://localhost:8080/actuator/configprops",
      "templated": false
    },
    "env": {
      "href": "http://localhost:8080/actuator/env",
      "templated": false
    },
    "env-toMatch": {
      "href": "http://localhost:8080/actuator/env/{toMatch}",
      "templated": true
    },
    "loggers": {
      "href": "http://localhost:8080/actuator/loggers",
      "templated": false
    },
    "loggers-name": {
      "href": "http://localhost:8080/actuator/loggers/{name}",
      "templated": true
    },
    "heapdump": {
      "href": "http://localhost:8080/actuator/heapdump",
      "templated": false
    },
    "threaddump": {
      "href": "http://localhost:8080/actuator/threaddump",
      "templated": false
    },
    "metrics": {
      "href": "http://localhost:8080/actuator/metrics",
      "templated": false
    },
    "metrics-requiredMetricName": {
      "href": "http://localhost:8080/actuator/metrics/{requiredMetricName}",
      "templated": true
    },
    "scheduledtasks": {
      "href": "http://localhost:8080/actuator/scheduledtasks",
      "templated": false
    },
    "mappings": {
      "href": "http://localhost:8080/actuator/mappings",
      "templated": false
    }
  }
}
```

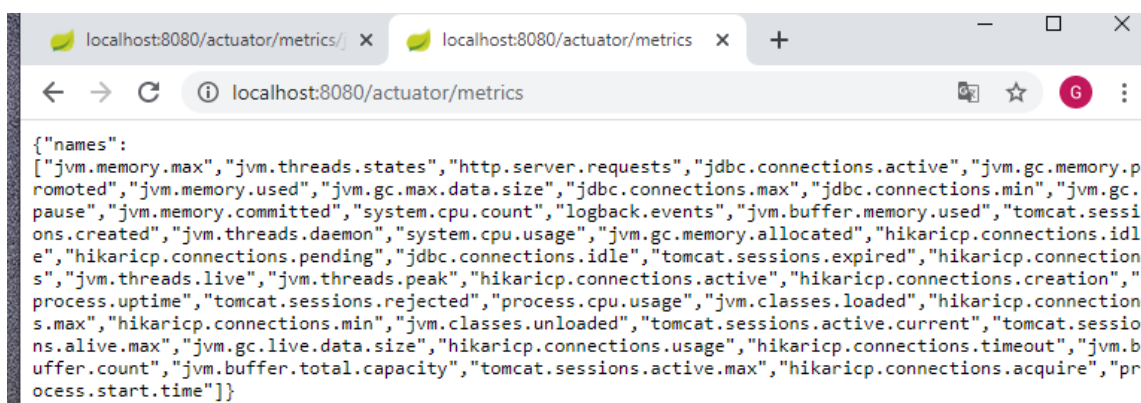
► On peut limiter aux endpoint voulus en modifiant l'instruction précédente dans le fichier `application.properties` pour **exposer seulement**, par exemple, les **endpoint health, info et metrics**:

`management.endpoints.web.exposure.include=health,info,metrics`

Le endpoint /metrics

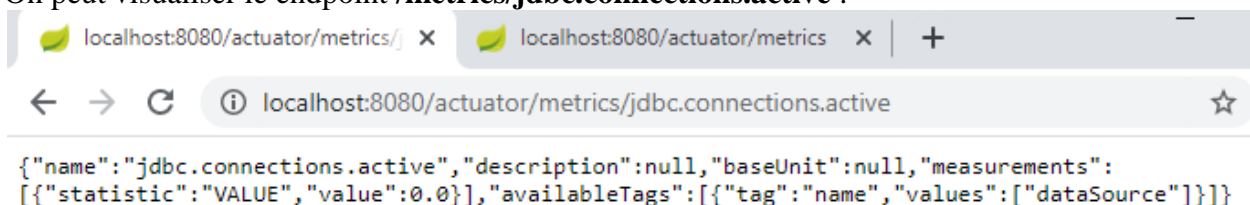
Cet endpoint donne des informations (mesures) sur les paramètres de l'application: mémoires utilisées, mémoire de la JVM utilisée, fichiers ouverts,, données sur tomcat, état de la base de données, URI appelés et leurs statistiques, etc.

Si on accède à <http://localhost:8080/actuator/metrics/>, on obtient la liste de tous les "sous-endpoints" qui donnent chacun accès à une série d'informations spécifiques :



```
{ "names": [ "jvm.memory.max", "jvm.threads.states", "http.server.requests", "jdbc.connections.active", "jvm.gc.memory.promoted", "jvm.memory.used", "jvm.gc.max.data.size", "jdbc.connections.max", "jdbc.connections.min", "jvm.gc.pause", "jvm.memory.committed", "system.cpu.count", "logback.events", "jvm.buffer.memory.used", "tomcat.sessions.created", "jvm.threads.daemon", "system.cpu.usage", "jvm.gc.memory.allocated", "hikaricp.connections.idle", "hikaricp.connections.pending", "jdbc.connections.idle", "tomcat.sessions.expired", "hikaricp.connections", "jvm.threads.live", "jvm.threads.peak", "hikaricp.connections.active", "hikaricp.connections.creation", "process.uptime", "tomcat.sessions.rejected", "process.cpu.usage", "jvm.classes.loaded", "hikaricp.connections.max", "hikaricp.connections.min", "jvm.classes.unloaded", "tomcat.sessions.active.current", "tomcat.sessions.alive.max", "jvm.gc.live.data.size", "hikaricp.connections.usage", "hikaricp.connections.timeout", "jvm.buffer.count", "jvm.buffer.total.capacity", "tomcat.sessions.active.max", "hikaricp.connections.acquire", "process.start.time" ] }
```

On peut visualiser le endpoint `/metrics/jdbc.connections.active` :



```
{ "name": "jdbc.connections.active", "description": null, "baseUnit": null, "measurements": [ { "statistic": "VALUE", "value": 0.0 }, { "availableTags": [ { "tag": "name", "values": [ "dataSource" ] } ] } ] }
```

Le endpoint /env

Liste toutes les variables d'environnement.

Le endpoint /beans

Liste toutes les beans créés par la BeanFactory.

On peut retrouver les divers beans de notre application en faisant une recherche sur la page avec le nom de la classe du bean recherché, par exemple le bean `HealthCheck`.

Créer un « metrics » en utilisant un compteur de visite d'une page

L'interface `io.micrometer.core.instrument.Meter` propose des sous interfaces pour réaliser des mesures : **Counter**, **Gauge**, **Timer** et d'autres.

On se propose de modifier le contrôleur REST pour ajouter un compteur qui compte le nombre d'accès au **endpoint /all**.

► **Modifier la classe du contrôleur en ajoutant les instructions écrites en rouge.**

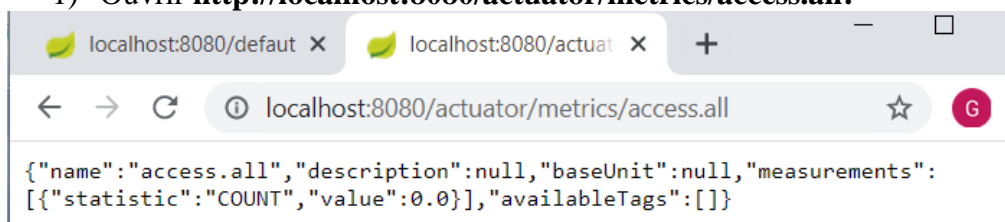
```
import io.micrometer.core.instrument.Counter;
import io.micrometer.core.instrument.MeterRegistry;

@RestController
public class InvitesController {
    @Autowired // Injection du repository
    private InviteRepository inviteRepository;
    ///////////////////////////////////////////////////
    Counter compteur; //le compteur
    public InvitesController(MeterRegistry registry) {
        compteur = registry.counter("access.all"); //enregistrement du compteur
    }
    ///////////////////////////////////////////////////
    -----
    @RequestMapping("/all")
    ArrayList<Invite> getAll() {
        compteur.increment();
        return (ArrayList<Invite>) inviteRepository.findAll();
    }
    -----
}
```

Le constructeur **InvitesController** enregistre le compteur qui est incrémenté à chaque accès au endpoint **/all**.

Cette métrique est accessible au endpoint **http://localhost:8080/actuator/metrics/access.all**

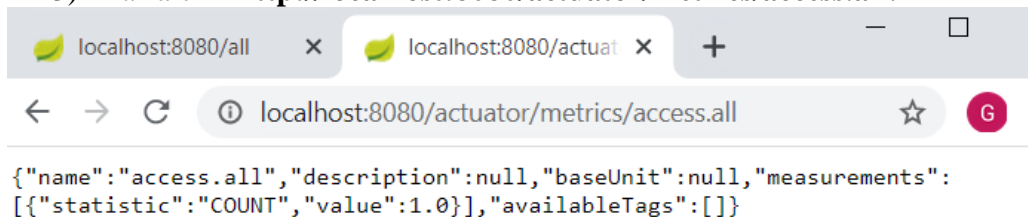
1) Ouvrir **http://localhost:8080/actuator/metrics/access.all**:



Le compteur vaut 0.

2) Ouvrir **http://localhost:8080/all**.

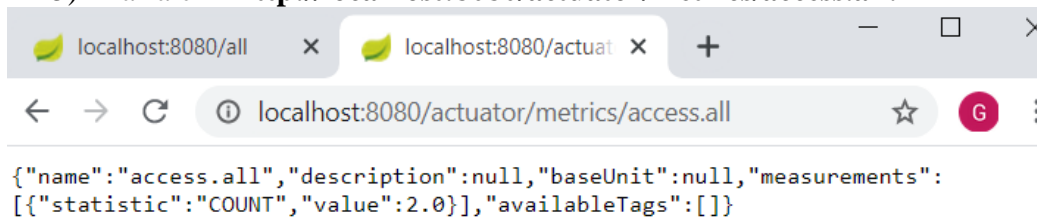
3) Rafraichir **http://localhost:8080/actuator/metrics/access.all**:



Le compteur vaut 1.

4) Rafraichir **http://localhost:8080/all**.

5) Rafraichir **http://localhost:8080/actuator/metrics/access.all:**



Le compteur vaut 2.

Et ainsi de suite.

Voir la documentation suivante

<https://docs.spring.io/spring-metrics/docs/current/public/atlas>
pour des explications et des exemples pour réaliser d'autres mesures.

Création du jar exécutable : le microservice

Le jar exécutable généré contient le serveur tomcat et l'application développée. Le serveur tomcat est dit embarqué. Le jar ainsi réalisé constitue un **microservice**.

Génération du jar

Clic droit sur le projet > Run As > Maven install

Ouvrir une invite de commande, aller dans le workspace puis dans le dossier du projet, se déplacer ensuite dans le dossier **target**, le **fichier nomDuProjet-version-SNAPSHOT.jar** a été créé.

Dans une invite de commande, se déplacer dans le dossier "**target**", exécuter l'application par la commande

java -jar nomDuProjet-version-SNAPSHOT.jar

Faire dans l'invite de commande **Ctrl c** 2 fois pour arrêter l'application.