

Atelier-Création de Microservice**Gestion des candidats****Objectifs**

Notre objectif est de créer un Microservice qui fournit des exemples d'opérations.

Développement des micro-services

Nous visons à créer un premier microservice de gestion des candidats dans une entreprise.

Soit l'entité suivante représentant un candidat:

Candidat
-ID: Integer -nom:String -prenom:String - email: String

Le microservice Gestion candidats va se charger des fonctionnalités suivantes :

- Afficher tous les candidats
- Afficher un candidat par son id ou son nom
- Ajouter/modifier/supprimer un candidat

1. Créez un projet **de type Spring boot** sur l'IDE **IntelliJ** :

Server URL: start.spring.io 

Name:

Location:
Project will be created in: ~\Downloads\MS_Reforme_Cours_Maroua\Works

Create Git repository

Language: Java Kotlin Groovy

Type: Gradle - Groovy Gradle - Kotlin Maven

Group:

Artifact:

Package name:

JDK:

Java:

Packaging: Jar War

 Next Cancel

2. Ajouter les dépendances nécessaires :

New Project

Spring Boot: 3.3.1

Dependencies:

Q rest X

- Developer Tools
 - Spring Boot DevTools
- Web
 - Spring Web
 - Rest Repositories
 - Rest Repositories HAL Explorer
 - Spring HATEOAS
 - Jersey
- NoSQL
 - Spring Data Elasticsearch (Access+Driver)
- Testing
 - Spring REST Docs
 - Contract Stub Runner
- Spring Cloud Discovery
 - Eureka Discovery Client
- Spring Cloud Routing
 - OpenFeign

Rest Repositories

Exposing Spring Data repositories over REST via Spring Data REST.

[Accessing JPA Data with REST](#) ↗

[Accessing Neo4j Data with REST](#) ↗

[Accessing MongoDB Data with REST](#) ↗

Added dependencies:

- ✗ Spring Boot Actuator
- ✗ Spring Data JPA
- ✗ H2 Database
- ✗ Config Client
- ✗ Eureka Discovery Client
- ✗ Spring Web
- ✗ Rest Repositories



Previous

Create

Cancel

3. Créez le contrôleur REST de notre microservice qui est représenté par une classe nommée par exemple “CandidatRestAPI”.

```
@RestController
public class CandidatRestAPI {
    private String title="Hello, i'm the candidate Micro-Service";

    @RequestMapping("/hello")
    public String sayHello(){
        System.out.println(title);
        return title;
    }

}
```

4. Créez l'entité "Candidat" :

```
@Entity
public class Candidat implements Serializable{
    private static final long serialVersionUID = 6

    @Id
    @GeneratedValue
    private int id;
    private String nom, prenom,email;
    public int getId() {
        return id;
    }
    public String getNom() {
        return nom;
    }
    public void setNom(String nom) {
        this.nom = nom;
    }
    public String getPrenom() {
        return prenom;
    }
    public void setPrenom(String prenom) {
        this.prenom = prenom;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public Candidat() {
        super();
        // TODO Auto-generated constructor stub
    }
    public Candidat(String nom) {
        super();
        this.nom = nom;
    }
}
```

5. Créez le Repository lié à l'entité Candidat :

```
public interface CandidatRepository extends JpaRepository<Candidat , Integer> {

}
```

6. Créez une fonction dans la classe CandidateApplication (créeée par défaut lors de la création du projet) qui va permettre d'insérer automatiquement des candidats dans la base

```

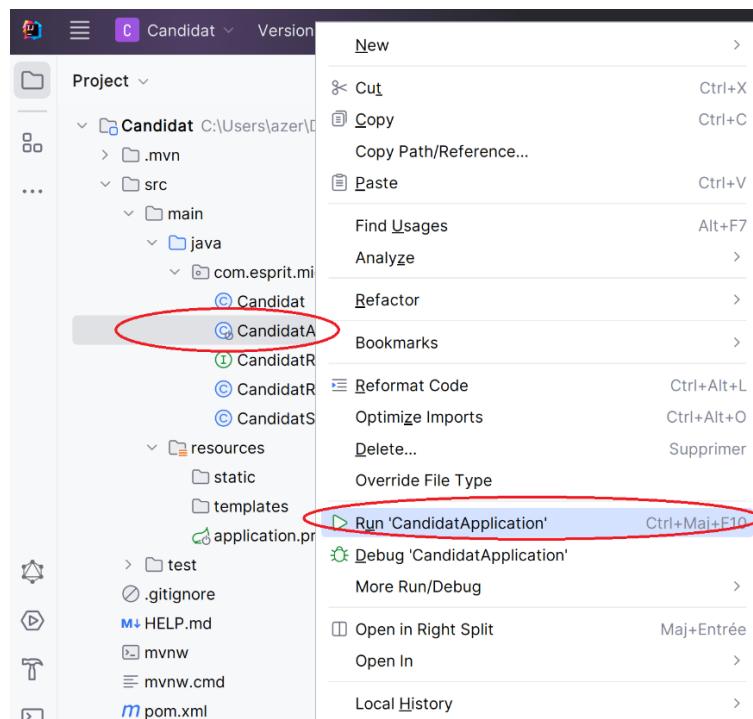
@SpringBootApplication
public class CandidateApplication {
    public static void main(String[] args) {
        SpringApplication.run(CandidateApplication.class, args);
    }
    @Autowired
    private CandidatRepository repository;

    @Bean
    ApplicationRunner init() {
        return (args) -> {
            // save
            repository.save(new Candidat("Mariem", "Ch", "ma@esprit.tn"));
            repository.save(new Candidat("Sarra", "ab", "sa@esprit.tn"));
            repository.save(new Candidat("Mohamed", "ba", "mo@esprit.tn"));
            repository.save(new Candidat("Maroua", "dh", "maroua@esprit.tn"));

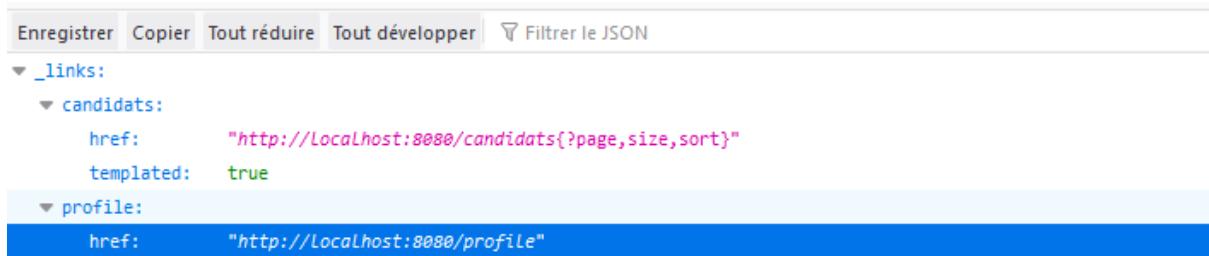
            // fetch
            repository.findAll().forEach(System.out::println);
        };
    }
}

```

7. Exécuter la classe CandidateApplication :



8. Accéder à l'adresse <http://localhost:8080/>



```
Enregistrer Copier Tout réduire Tout développer Filtrer le JSON
▼ _links:
  ▼ candidats:
    href: "http://localhost:8080/candidats{?page,size,sort}"
    templated: true
  ▼ profile:
    href: "http://localhost:8080/profile"
```

Vous pouvez accéder à l'adresse <http://localhost:8080/candidats> pour voir les détails de tous les candidats.

Vous pouvez également faire une recherche de candidats par id : <http://localhost:8080/candidats/4>

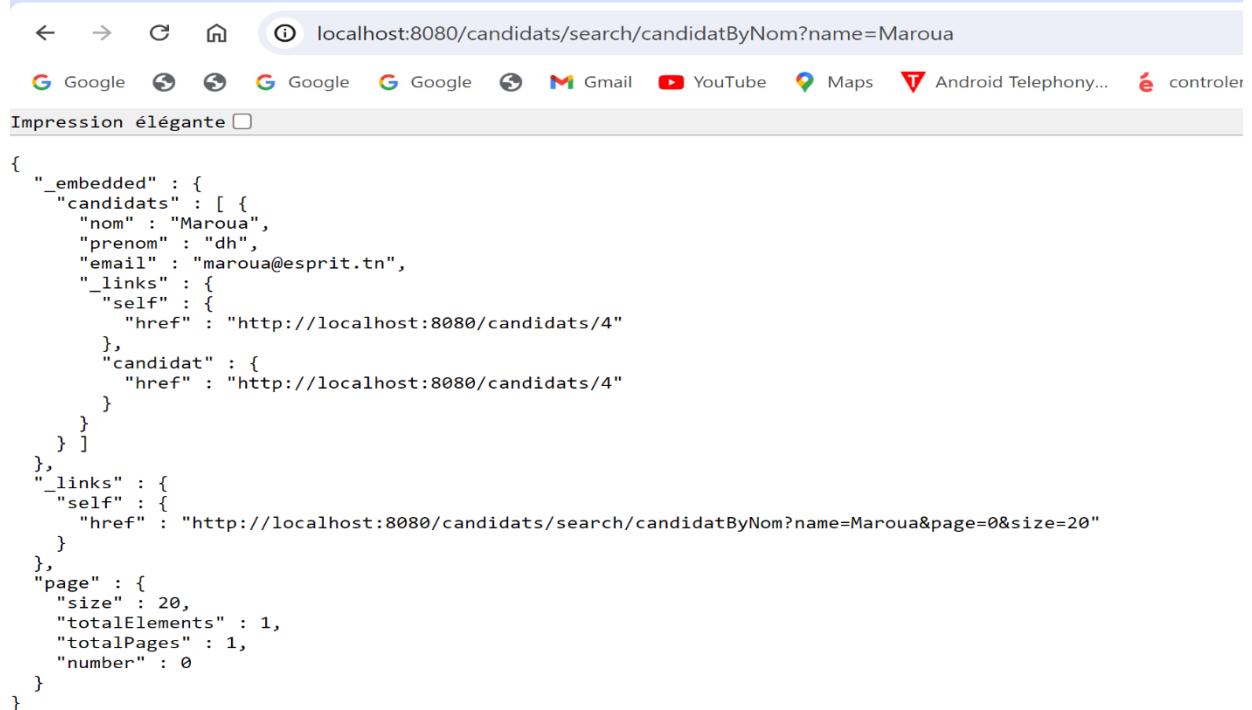
9. Pour ajouter par exemple une requête de recherche d'un candidat par son nom, on modifie ainsi la classe CandidatRepository:

```
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;

public interface CandidatRepository extends JpaRepository<Candidat, Integer> {

    @Query("select c from Candidat c where c.nom like :name")
    public Page<Candidat> candidatByNom(@Param("name") String n, Pageable pageable);
}
```

Lancez l'URL <http://localhost:8080/candidats/search/candidatByNom?name=Maroua> pour faire la recherche d'un candidat par nom



localhost:8080/candidats/search/candidatByNom?name=Maroua

```
{
  "_embedded": {
    "candidats": [
      {
        "nom": "Maroua",
        "prenom": "dh",
        "email": "maroua@esprit.tn",
        "_links": {
          "self": {
            "href": "http://localhost:8080/candidats/4"
          },
          "candidat": {
            "href": "http://localhost:8080/candidats/4"
          }
        }
      }
    ]
  },
  "_links": {
    "self": {
      "href": "http://localhost:8080/candidats/search/candidatByNom?name=Maroua&page=0&size=20"
    }
  },
  "page": {
    "size": 20,
    "totalElements": 1,
    "totalPages": 1,
    "number": 0
  }
}
```

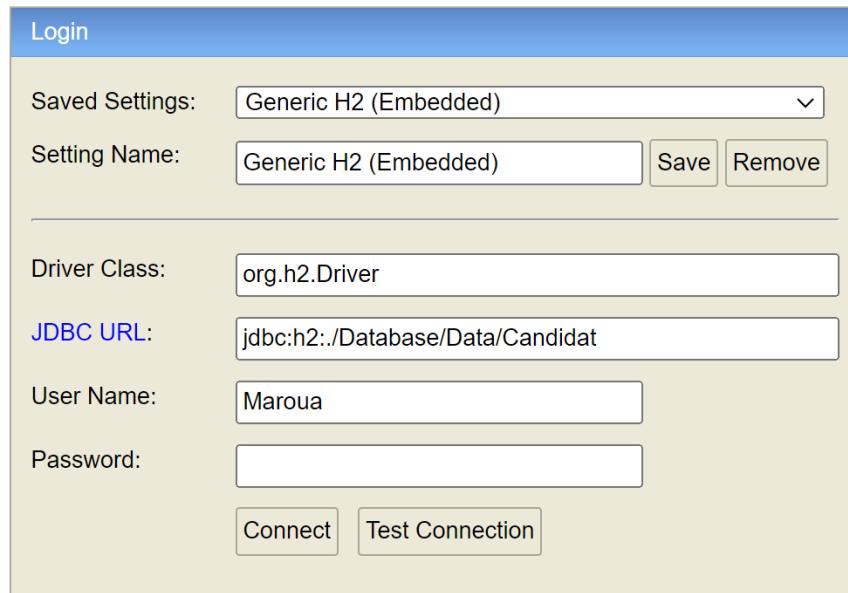
9. Pour le monitoring de la base de données H2, ajoutez la configuration suivante dans le fichier **application.properties**:

```
# H2
spring.h2.console.enabled=true
spring.h2.console.path=/h2

# Datasource
spring.datasource.username=Maroua
spring.datasource.password=
spring.datasource.url=jdbc:h2:file:./Database/Data/Candidat
spring.datasource.driver-class-name=org.h2.Driver
spring.jpa.hibernate.ddl-auto = create
```

Accédez ensuite à l'URL <http://localhost:8080/h2> pour consulter l'interface de gestion de la base H2.

Ajouter l'URL d'accès au JDBC **./Database/Data/Candidat** pour voir et traiter les données de votre BD h2. Donner votre userName et password :



The screenshot shows the H2 DBMS interface. On the left, there's a tree view of the database schema with nodes for CANDIDAT, INFORMATION_SCHEMA, and SEQUENCES. The CANDIDAT node has child nodes for ID, EMAIL, NOM, and PRENOM. The INFORMATION_SCHEMA node has child nodes for Indexes, Sequences, and Users. Below the schema tree, there's a code editor window showing Java code for a service class named CandidatService. The code includes imports for org.springframework.stereotype.Service and org.springframework.transaction.annotation.Transactional, and annotations for @Service and @Transactional. The code also includes a constructor injection for a CandidatRepository and a method named saveCandidate. The right side of the interface shows the results of a SQL query: 'SELECT * FROM CANDIDAT;'. The results are displayed in a table with columns ID, EMAIL, NOM, and PRENOM. The data shows four rows of candidates: 1 (ma@esprit.tn, Mariem, Ch), 2 (sa@esprit.tn, Sarra, ab), 3 (mo@esprit.tn, Mohamed, ba), and 4 (maroua@esprit.tn, Maroua, dh).

```
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
@Service
public class CandidatService {
    @Autowired
    private CandidatRepository candidatRepository;
    @Transactional
    public void saveCandidate(Candidat candidat) {
        candidatRepository.save(candidat);
    }
}
```

ID	EMAIL	NOM	PRENOM
1	ma@esprit.tn	Mariem	Ch
2	sa@esprit.tn	Sarra	ab
3	mo@esprit.tn	Mohamed	ba
4	maroua@esprit.tn	Maroua	dh

10. Nous allons maintenant préparer les services “ajouter”, “modifier” et “supprimer” candidat.

Ajoutez la classe CandidatService qui va se charger d'implémenter ces méthodes:

```

public Candidat updateCandidat(int id, Candidat newCandidat) {
    if (candidateRepository.findById(id).isPresent()) {

        Candidat existingCandidat = candidateRepository.findById(id).get();
        existingCandidat.setNom(newCandidat.getNom());
        existingCandidat.setPrenom(newCandidat.getPrenom());
        existingCandidat.setEmail(newCandidat.getEmail());

        return candidateRepository.save(existingCandidat);
    } else
        return null;
}

public String deleteCandidat(int id) {
    if (candidateRepository.findById(id).isPresent()) {
        candidateRepository.deleteById(id);
        return "candidat supprimé";
    } else
        return "candidat non supprimé";
}
}

```

Nous préparons par la suite notre API REST. Configurer les requêtes REST dans le contrôleur REST en ajoutant les méthodes qui suivent:

```

@.Autowired
private CandidatService candidatService;

@PostMapping(consumes = MediaType.APPLICATION_XML_VALUE)
@ResponseStatus(HttpStatus.CREATED)
public ResponseEntity<Candidat> createCandidat(@RequestBody Candidat candidat) {
    return new ResponseEntity<>(candidatService.addCandidat(candidat), HttpStatus.OK);
}

@PutMapping(value = "/{id}", produces = MediaType.APPLICATION_JSON_VALUE)
@ResponseStatus(HttpStatus.OK)
public ResponseEntity<Candidat> updateCandidat(@PathVariable(value = "id") int id,
                                                @RequestBody Candidat candidat) {
    return new ResponseEntity<>(candidatService.updateCandidat(id, candidat),
                                HttpStatus.OK);
}

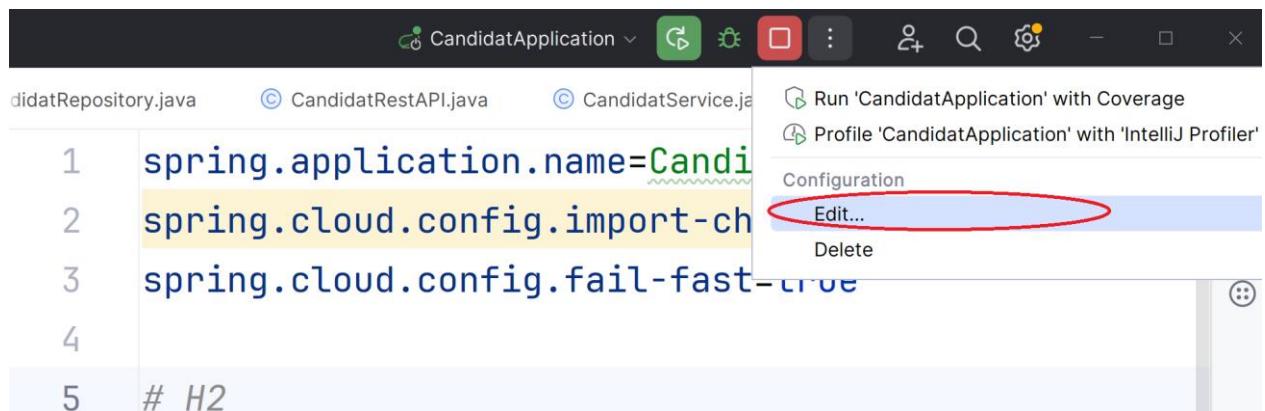
@DeleteMapping(value = "/{id}", produces = MediaType.APPLICATION_JSON_VALUE)
@ResponseStatus(HttpStatus.OK)
public ResponseEntity<String> deleteCandidat(@PathVariable(value = "id") int id) {
    return new ResponseEntity<>(candidatService.deleteCandidat(id), HttpStatus.OK);
}

```

11. Afin de lancer une deuxième instance du même service sur un port différent :

❖ **IDE IntelliJ** :

- accédez à **Edit Configuration** :

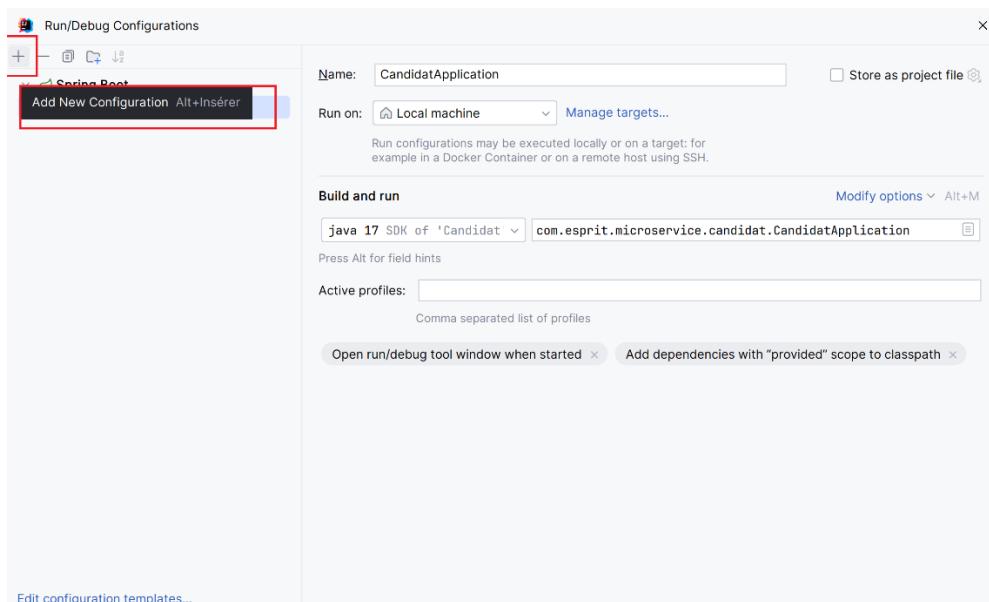


A screenshot of the IntelliJ IDEA interface. The code editor shows a configuration file with the following content:

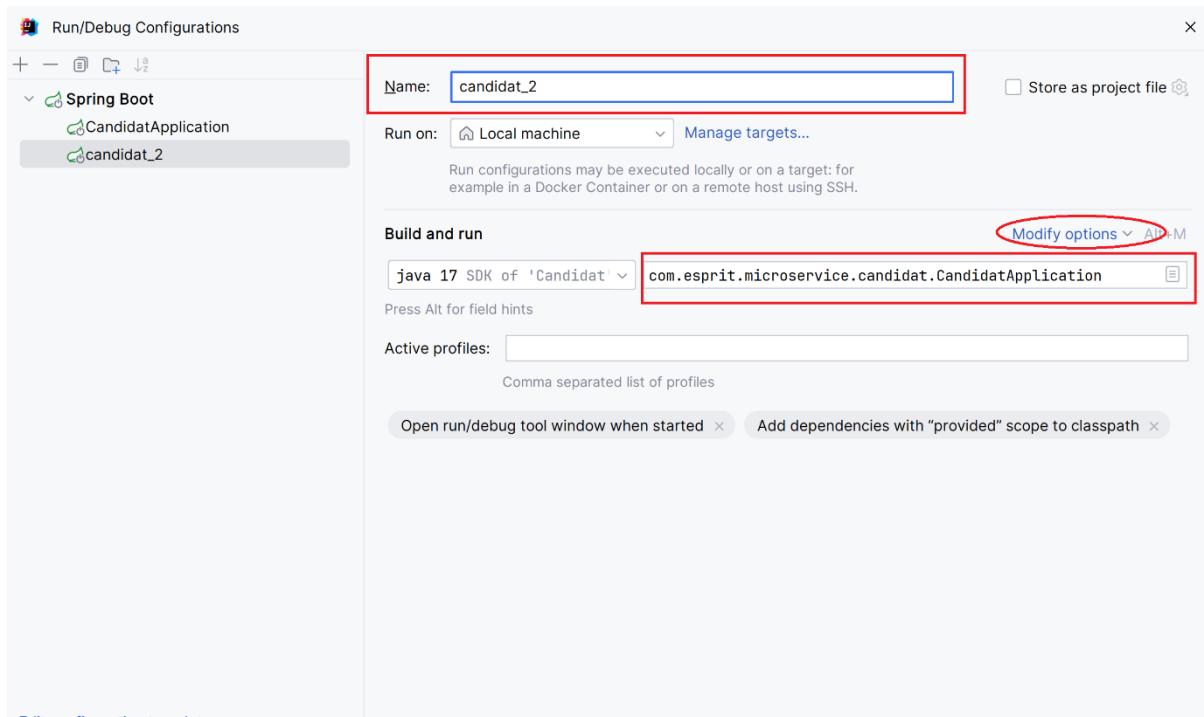
```
1 spring.application.name=Candidat
2 spring.cloud.config.import-ch
3 spring.cloud.config.fail-fast=true
4
5 # H2
```

The context menu is open over the line "spring.cloud.config.import-ch". The "Edit..." option is highlighted with a red circle.

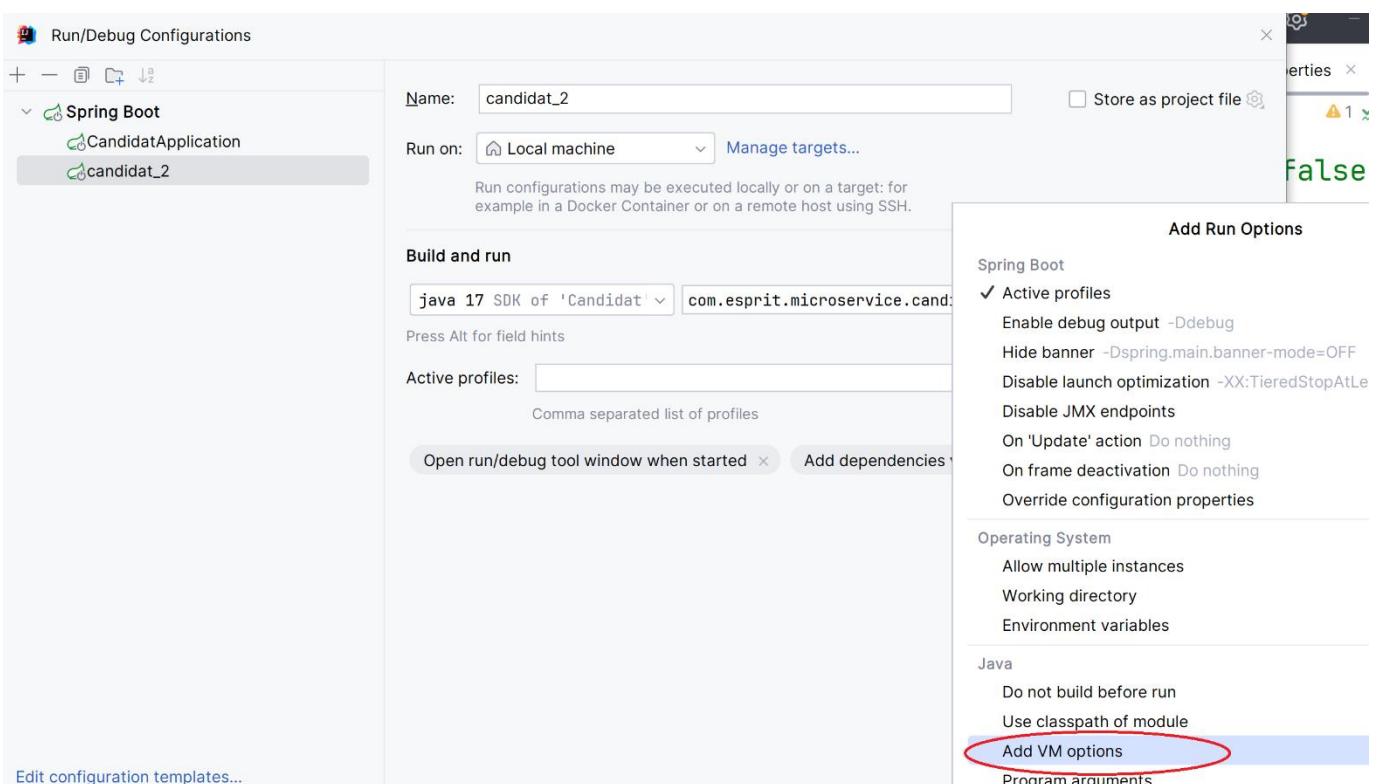
- Cliquez sur « Add New Configuration » :



- Choisissez « Spring Boot » puis ajouter la configuration suivante en choisissant la classe CandidateApplication comme « Main class ».

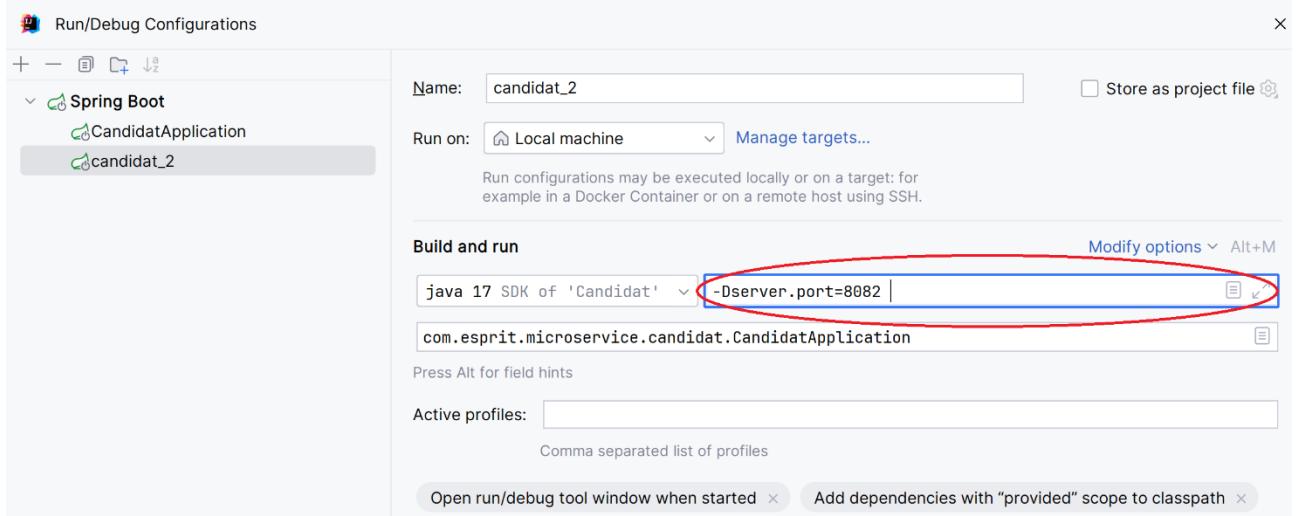


d. Cliquez sur « Modify options » --> « Add VM options » :



e. Fixez le port de la nouvelle instance, du même service, en ajoutant l'argument :

-Dserver.port=8082 :



12. Exécutez la nouvelle instance et y accéder à travers l'URL « <http://localhost:8082/>

** Ne pas arrêter l'ancienne instance du service s'exécutant sur le port 8080.

13. Travail à faire :

Implémentez un deuxième microservice (dans un nouveau projet spring boot) qui se charge de la gestion des jobs et utilisant MySQL comme Base de données. L'entité Job est représentée comme suit:

Job
-ID: Integer
-Service: String
-Etat: Boolean

Ce deuxième microservice prendra en charge les fonctionnalités suivantes:

- Afficher tous les jobs
- Afficher un job par son id ou son nom
- La modification de l'état de poste :
 - o Etat= oui (si poste disponible)
 - o Etat= non (si poste est occupé).