TP N°2: Développement d'une WebApp et communication avec un Microservice via Rest API

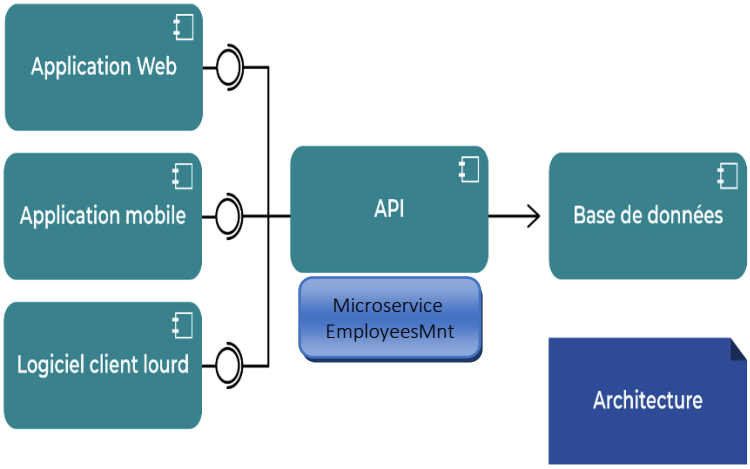
1. Prérequis
   - Eclipse Mars (ou autre 202X...) avec le plugin Maven 3.x ;
   - JDK 1.8;
   - Connection à Internet pour permettre à Maven de télécharger les dépendances nécessaires (Spring Boot 2.17, …).
   - POSTMAN ou un autre outil pour tester les méthodes POST, PUT et DELETE.

2. Objectifs
   - ✓ Développer une Webapp de Gestion des employés avec Spring Boot.
   - ✓ Développer un microservice Rest API avec Spring Boot qui offre les service CRUD pour les données des employés
   - ✓ Utilisation de la base de données de type H2
   - ✓ Communication de l'API avec la Base de données H2
   - ✓ Communication Rest API de la WebApp avec le microservice via le composant RestTemplate
   - ✓ Utilisation du starter Thymeleaf pour la gestion des composants de la Vue.

3. Mise en situation

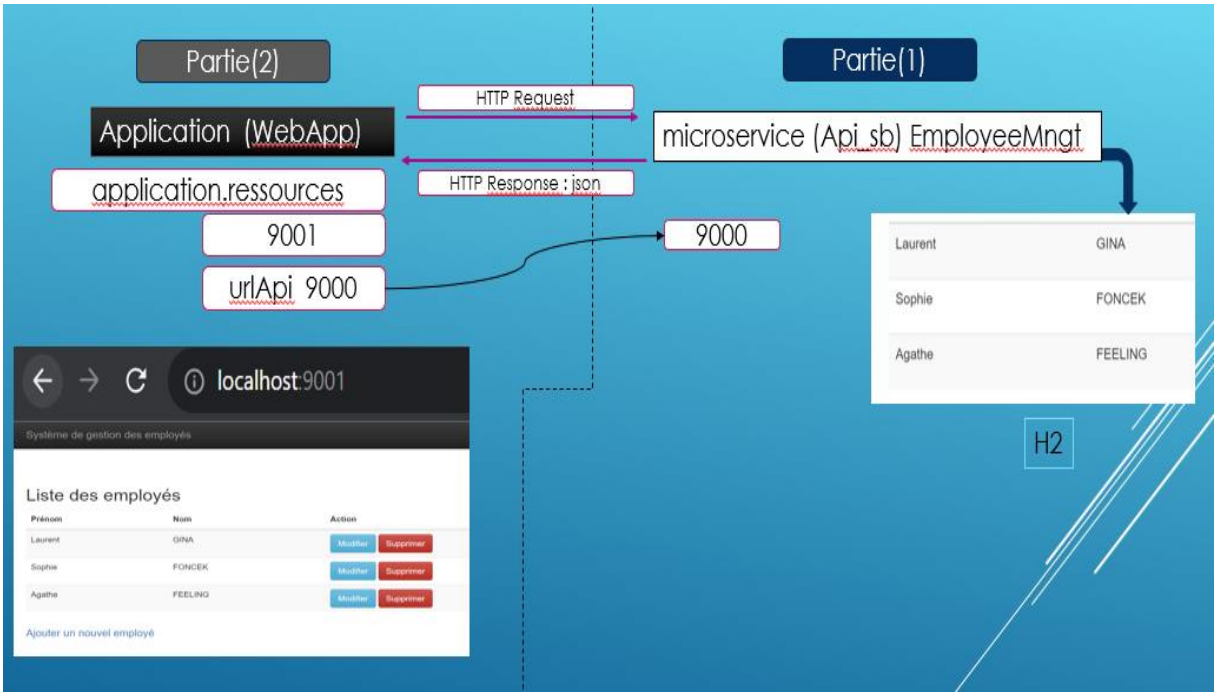
- ✓ Une entreprise « myHR » souhaite offrir un service de gestion d'employés aux petites entreprises.
- ✓ L'idée est d'offrir une suite d'outils (application web, application mobile) prête à l'emploi.
- ✓ Pour lancer ce projet, myHR souhaite avant tout mettre à disposition une API qui permettra à toutes les autres applications d'accéder aux mêmes données.
- ✓ L'idée étant de gérer des employés, l'API devra donc offrir un CRUD pour les données des employés.
- ✓ Les données seront dans une base de données H2 :
  - H2 est une base de données relationnelle Java très légère, qui par défaut fonctionne en "in memory".
  - Au démarrage du programme, la structure de la base est construite ;
  - Lorsque le programme s'arrête, le contenu de la base de données est supprimé.
- ✓ L'API Rest devra donc exposer des Endpoints correspondant aux actions du CRUD, et communiquer avec la base de données pour récupérer ou modifier les informations des employés.

4. Architecture de mise en œuvre



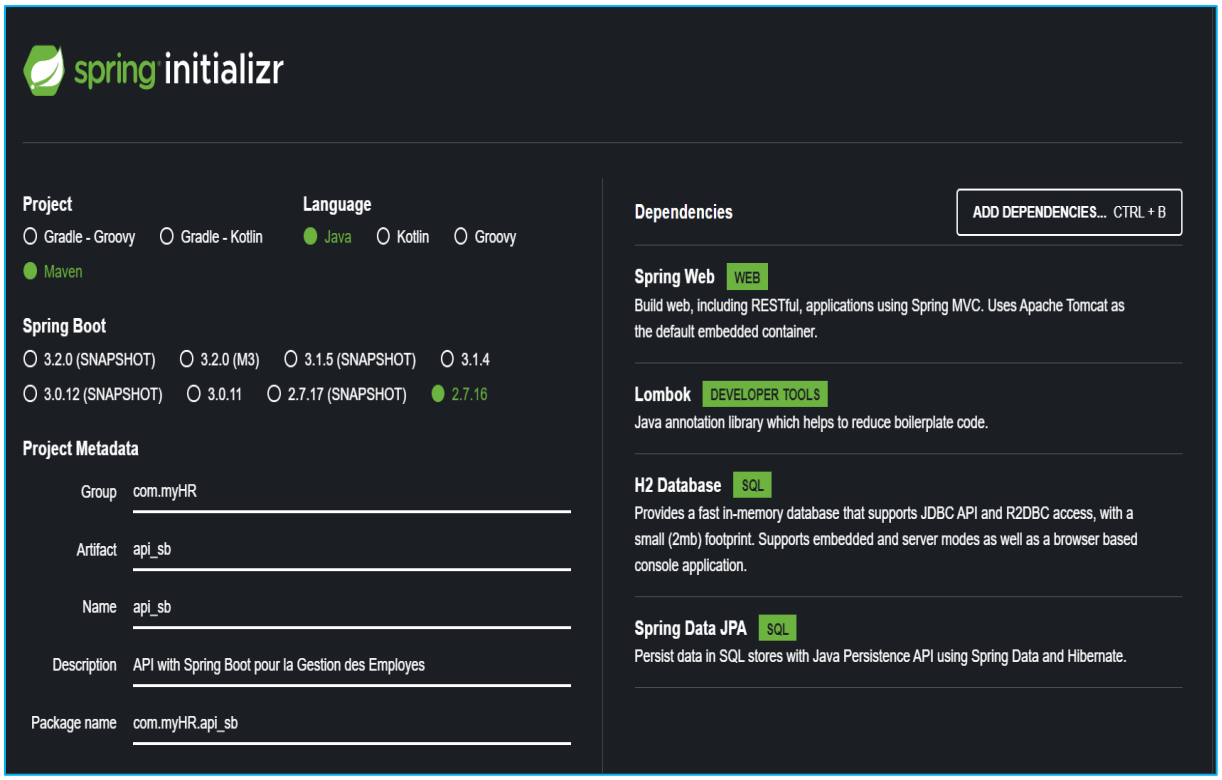
5. Démarche de développement de l'application :

1. Partie (1) → Développement du microservice : EmployeesMngt
2. Partie (2)→ Développement de la WebApp
3. Faire communiquer les deux Composants



6. Mise en œuvre de la ‘API

a. https://start.spring.io/



- Pour les « Project Metadata » :
  - group: com.myHR
  - artifact: api_SB
- Pour les dépendances :
  - **Spring Web** : permet de faire du RESTful, ce qui correspond à notre API pour exposer des endpoints.
  - **Lombok** : une librarie pour optimiser certaines classes,
  - **H2 Database** : une base de données In-Memory.
  - **Spring Data JPA** : permet de gérer la persistance des données avec la base de données

➔ Le fichier pom.xml généré

```
<?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>2.7.16</version>
```

```
		<relativePath/> <!-- lookup parent from repository -->
	</parent>
	<groupId>com.myHR</groupId>
	<artifactId>api_sb</artifactId>
	<version>0.0.1-SNAPSHOT</version>
	<name>api_sb</name>
	<description>API with Spring Boot</description>
	<properties>
		<java.version>1.8</java.version>
	</properties>
	<dependencies>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-data-jpa</artifactId>
		</dependency>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-web</artifactId>
		</dependency>

		<dependency>
			<groupId>com.h2database</groupId>
			<artifactId>h2</artifactId>
			<scope>runtime</scope>
		</dependency>
		<dependency>
			<groupId>org.projectlombok</groupId>
			<artifactId>lombok</artifactId>
			<optional>true</optional>
		</dependency>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-test</artifactId>
			<scope>test</scope>
		</dependency>
	</dependencies>

	<build>
		<plugins>
			<plugin>
				<groupId>org.springframework.boot</groupId>
				<artifactId>spring-boot-maven-plugin</artifactId>
				<configuration>
					<excludes>
						<exclude>

	<groupId>org.projectlombok</groupId>
							<artifactId>lombok</artifactId>
						</exclude>
					</excludes>
				</configuration>
			</plugin>
		</plugins>
	</build>

</project>
```

➔ En résumé :

Pour implémenter une API qui communique avec une base de données, 3 éléments sont essentiels :

- **Le starter web** qui permettra d'exposer les endpoints.
- Un starter pour **gérer la persistance** des données (comme Spring Data JPA).
- **La dépendance pour le driver** de la base de données concernée (par ex. H2 Database ou MySQL Driver).

7. Configuration de la base de données

A. Pour configurer la base de données H2, il existe plusieurs méthodes possibles, Cependant, il est conseillé de laisser le comportement par défaut qui implique zéro configuration, et à ajouter uniquement la propriété pour activer la console de visualisation de la base de données.

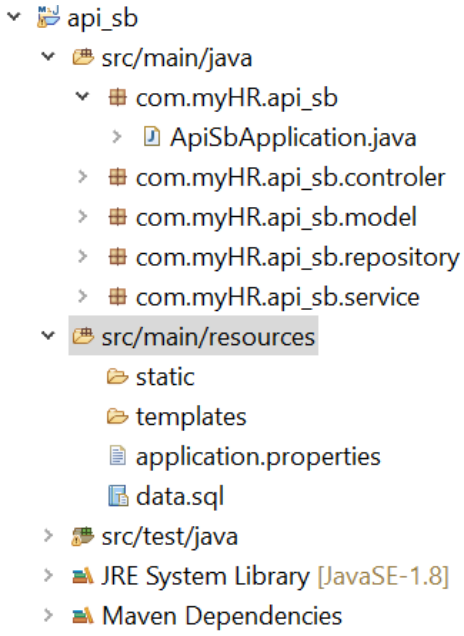
B. Pour gérer les données au sein de la base de données, il est possible de de fournir la structure de la base de données et des données.

C. Le fichier nommé « data.sql » ou « schema.sql » contient la structure qui sera utilisée, ainsi que quelques données.

   Il s'agit d'une unique table nommée Employees, avec 5 colonnes.

   Ce fichier est à placer dans le répertoire src/main/resources.

Ainsi et grâce à la puissance de Spring Boot : il sera pris en compte automatiquement sans que vous ayez quoi que ce soit à faire. De ce fait, le script SQL sera exécuté au démarrage de l'application, et la base de données contiendra la table et les données.



➔ application.properties

```
#Global configuration
spring.application.name=api_sb
#Tomcat configuration
server.port=9000
#Log level configuration
logging.level.root=ERROR
logging.level.com.myHR=INFO
logging.level.org.springframework.boot.autoconfigure.h2=INFO
logging.level.org.springframework.boot.web.embedded.tomcat=INFO
#H2 Configuration
spring.h2.console.enabled=true
```

spring.application.name=api : pour définir un nom à l'application ;

server.port=9000 : pour ne pas être sur le port par défaut 8080 ;

logging.level :

root=ERROR : par défaut, seules les traces en ERROR s'affichent.

L'idée est simple : réduire les affichages dans la console de toutes les "3rd party",

com.myHR =INFO : pour ce qui est de notre code, on est en INFO pour avoir du détail,

org.springframework.boot.autoconfigure.h2=INFO : permet de voir dans la console l'URL jdbc de la base H2,

org.springframework.boot.web.embedded.tomcat : permet de voir dans la console le port utilisé par Tomcat au démarrage ;

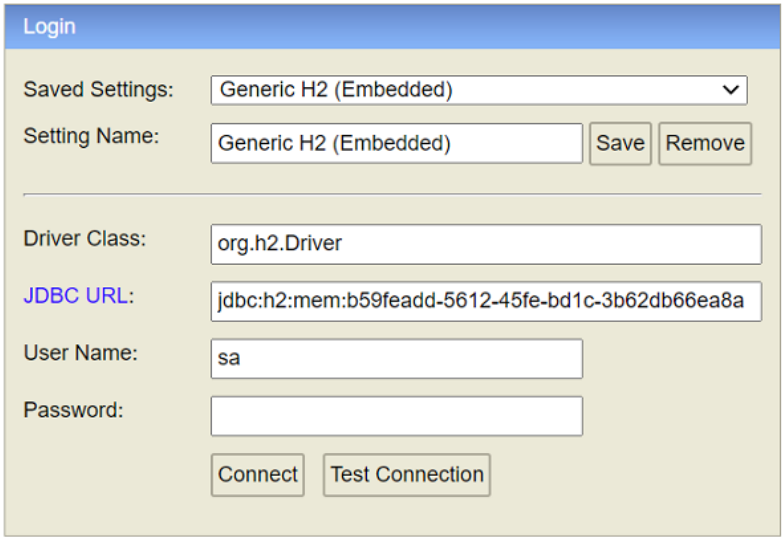
spring.h2.console.enabled=true : correspond à la propriété pour activité de la console H2.

Concernant la console H2, une fois l'application démarrée, vous pouvez aller sur l'URL "http://localhost:9000/h2-console". Une fenêtre de login s'ouvre, et il est nécessaire d'indiquer l'URL Jdbc (qui change à chaque démarrage de l'application).

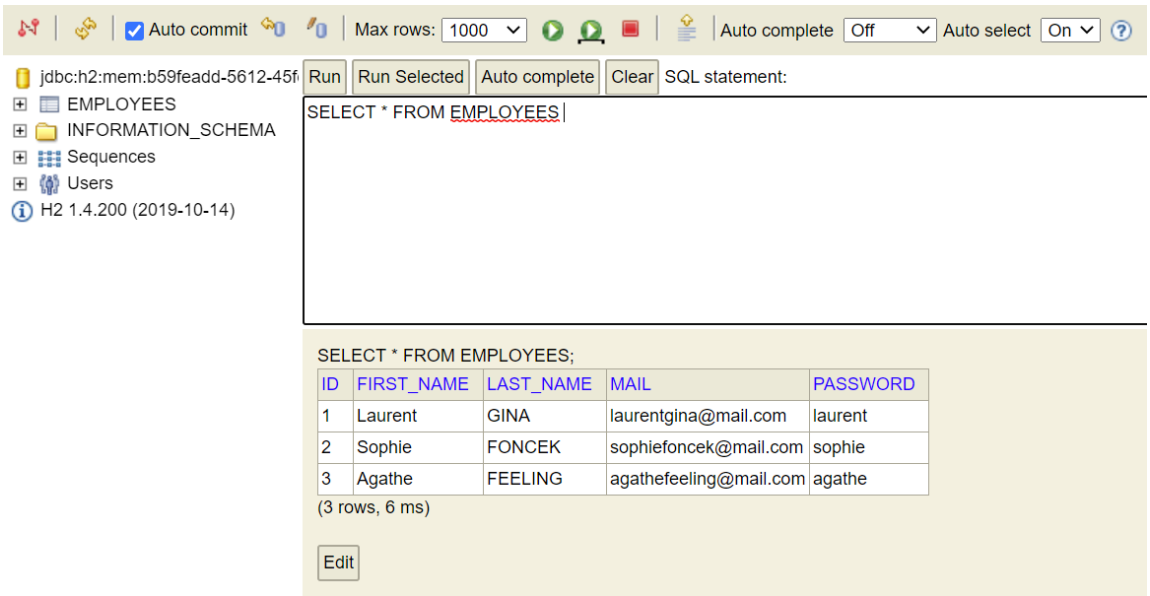
Dans la console de démarrage de l'application, une ligne doit ressembler à la suivante :

```
H2 console available at '/h2-console'. Database available at 'jdbc:h2:mem:b59feadd-5612-45fe-bd1c-3b62db66ea8a'
```

Récupérez l'URL JDBC (en l'occurrence jdbc:h2:mem:b59feadd-5612-45fe-bd1c-3b62db66ea8a'), saisissez dans le formulaire comme ci-dessous, puis "Connect". Le username par défaut est bien "sa", et le password par défaut est vide.



Une fois connecté, vous pouvez consulter le contenu de votre table :



➔ En résumé :

- La structure des packages reste le standard : controller / service / repository / model.
- Grâce à Spring Boot, la mise en œuvre de la base de données requiert 0 ligne de configuration, si ce n'est pour activer la console H2.
- Prochaines étapes : créer un contrôleur REST pour gérer les données.

8. Partie(1) Développement du microservice : EmployeesMngt

8.1 Implémentation du modèle

```
package com.myHR.api_sb.model;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

import lombok.Data;

@Data
@Entity
@Table(name = "employees")
public class Employee {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;

   @Column(name="first_name")
   private String firstName;

   @Column(name="last_name")
   private String lastName;

   private String mail;
   private String password;
}
```

- @Data est une annotation Lombok.
- @Entity est une annotation qui indique que la classe correspond à une table de la DB.
- @Table(name="employees") indique le nom de la table associée.
- L'attribut « id » correspond à la clé primaire de la table, et est donc annoté @Id.
- l'id est auto-incrémenté, on a ajouté l'annotation @GeneratedValue(strategy = GenerationType.IDENTITY).

- firstName et lastName sont annotés avec @Column pour faire le lien avec le nom du champ de la table.

8.2 Communication avec la base de données

- La communication entre l'applicatif et la base de données peut être fait via une Interface !
- En effet, la puissance du composant Spring Data JPA nous permet d'exécuter des requêtes SQL, sans avoir besoin de les écrire.
- Dans le package `com.myHR.api_sb.repository`, créez une interface nommée EmployeeRepository :

```
package com.myHR.api_sb.repository;
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;
import com.myHR.api_sb.model.Employee;
//@Repository est une annotation Spring pour indiquer que la classe est un bean,
// son rôle est de communiquer avec une source de données ( la base de données par exp).
//@Repository est une spécialisation de @Component.
//@Component, permet de déclarer auprès de Spring qu'une classe est un bean à exploiter.
@Repository
public interface EmployeeRepository extends CrudRepository<Employee, Long> {}
```

- CrudRepository : interface fournie par Spring. Elle fournit des méthodes pour manipuler l'entité.
- Utilise la généricité pour que son code soit applicable
- à n'importe quelle entité, d'où la syntaxe "CrudRepository<Employee, Long>" .
- La classe entité fournie doit être annotée @Entity, sinon Spring retournera une erreur.
- Ainsi, vous pouvez utiliser les méthodes définies par l'interface CrudRepository :

Modifier and Type	Method	Description
long	count()	Returns the number of entities available.
void	delete(T entity)	Deletes a given entity.
void	deleteAll()	Deletes all entities managed by the repository.
void	deleteAll(Iterable <? extends T> entities)	Deletes the given entities.
void	deleteAllById(Iterable <? extends ID> ids)	Deletes all instances of the type T with the given IDs.
void	deleteById(ID id)	Deletes the entity with the given id.
boolean	existsById(ID id)	Returns whether an entity with the given id exists.
Iterable <T>	findAll()	Returns all instances of the type.
Iterable <T>	findAllById(Iterable <ID> ids)	Returns all instances of the type T with the given IDs.
Optional <T>	findById(ID id)	Retrieves an entity by its id.
<S extends T> S	save(S entity)	Saves a given entity.
<S extends T> Iterable <S>	saveAll(Iterable <S> entities)	Saves all given entities.

8.3 Implémentation du service métier

Pour rappel :

Couche	Objectif
controller	Réceptionner la requête et fournir la réponse
service	Exécuter les traitements métiers
repository	Communiquer avec la source de données
model	Contenir les objets métiers

La couche service est également un pont entre le controller et le repository. De ce fait, nous allons créer une classe **EmployeeService**

```
package com.myHR.api_sb.service;
import java.util.Optional;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.myHR.api_sb.model.Employee;
import com.myHR.api_sb.repository.EmployeeRepository;
import lombok.Data;
//@Service : tout comme l'annotation @Repository, c'est une spécialisation de @Component.

//Son rôle est le même, mais son nom a une valeur sémantique pour ceux qui lisent le code.

@Data
@Service
```

```
public class EmployeeService {
   @Autowired
   private EmployeeRepository employeeRepository;
   public Optional<Employee> getEmployee(final Long id) {
      return employeeRepository.findById(id);
   }
   public Iterable<Employee> getEmployees() {
      return employeeRepository.findAll();
   }
   public void deleteEmployee(final Long id) {
      employeeRepository.deleteById(id);
   }
   public Employee saveEmployee(Employee employee) {
      Employee savedEmployee = employeeRepository.save(employee);
      return savedEmployee;
   }
}
```

8.4 Implémentation du controller : endpoint REST

- Un endpoint est associé à une URL. Lorsqu'on appelle cette URL, on reçoit une réponse, et cet échange se fait en HTTP.
- Un endpoint est une classe Java annotée **@RestController.**
- **L**es méthodes de la classe sont annotées : Chaque méthode annotée devient alors un endpoint grâce aux annotations ci-dessous :

Annotation	Type HTTP	Objectif
@GetMapping	GET	Pour la **lecture** de données.
@PostMapping	POST	Pour **l'envoi** de données. Cela sera utilisé par exemple pour créer un nouvel élément.
@PatchMapping	PATCH	Pour **la mise à jour partielle** de l'élément envoyé.
@PutMapping	PUT	Pour **le remplacement complet** de l'élément envoyé.
@DeleteMapping	DELETE	Pour la **suppression** de l'élément envoyé.
@RequestMapping		Intègre tous les types HTTP. Le type souhaité est indiqué comme attribut de l'annotation. Exemple : @RequestMapping(method = RequestMethod.GET)

```
package com.myHR.api_sb.controler;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import com.myHR.api_sb.model.Employee;
```

```
import com.myHR.api_sb.service.EmployeeService;

@RestController
public class EmployeeController {

   @Autowired
   private EmployeeService employeeService;

   /**
   * Read - Get all employees
   * @return - An Iterable object of Employee full filled
   */

   @GetMapping("/employees")
   public Iterable<Employee> getEmployees() {
      return employeeService.getEmployees();
   }
}
```

@RestController a 2 objectifs :

- Comme @Component, elle permet d'indiquer à Spring que cette classe est un bean.
  Elle indique à Spring d'insérer le retour de la méthode au format JSON dans le corps de la réponse HTTP. Grâce à ce deuxième point, les applications qui vont communiquer avec l'API accéderont au résultat de leur requête en parsant la réponse HTTP.
- Deuxièmement, L'injection dune instance d'EmployeeService. Cela permettra d'appeler les méthodes pour communiquer avec la base de données.

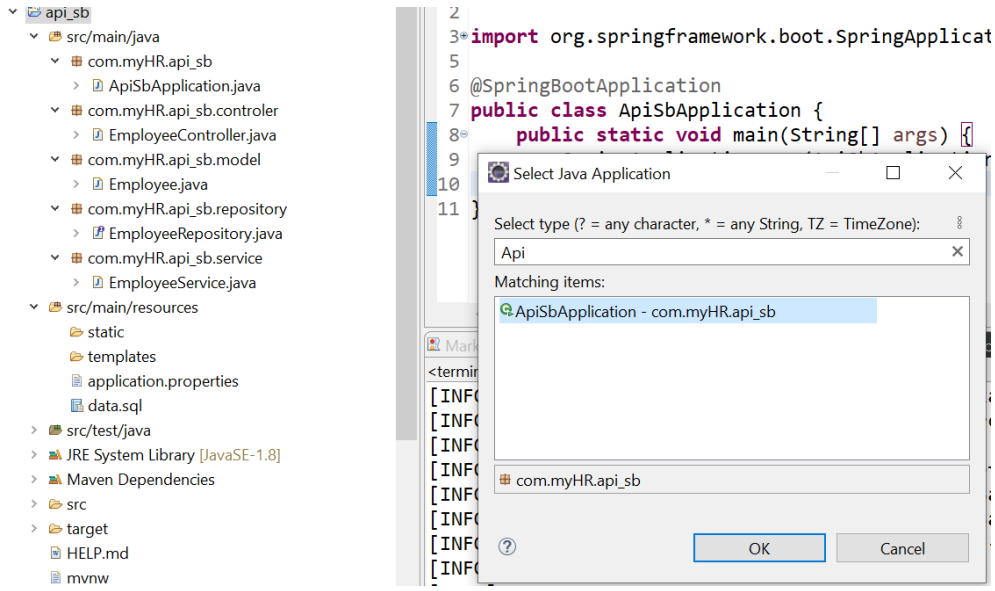
Par la suite, On a créé une méthode getEmployees() annotée @GetMapping("/employees") :

Cela signifie que les requêtes HTTP de type GET à l'URL /employees exécuteront le code de cette méthode: il s'agit d'appeler la méthode getEmployees() du service, ce dernier appellera la méthode findAll() du repository, et nous obtiendrons ainsi tous les employés enregistrés en base de données.

➔ En résumé :

- Notre entité du model est modélisée, et @Entity est l'annotation obligatoire.
- La communication aux données s'effectue via une classe annotée @Repository.
- La classe annotée @Service se charge des traitements métiers.
- Les controllers @RestController permettent de définir des URL et le code à exécuter quand ces dernières sont requêtées.

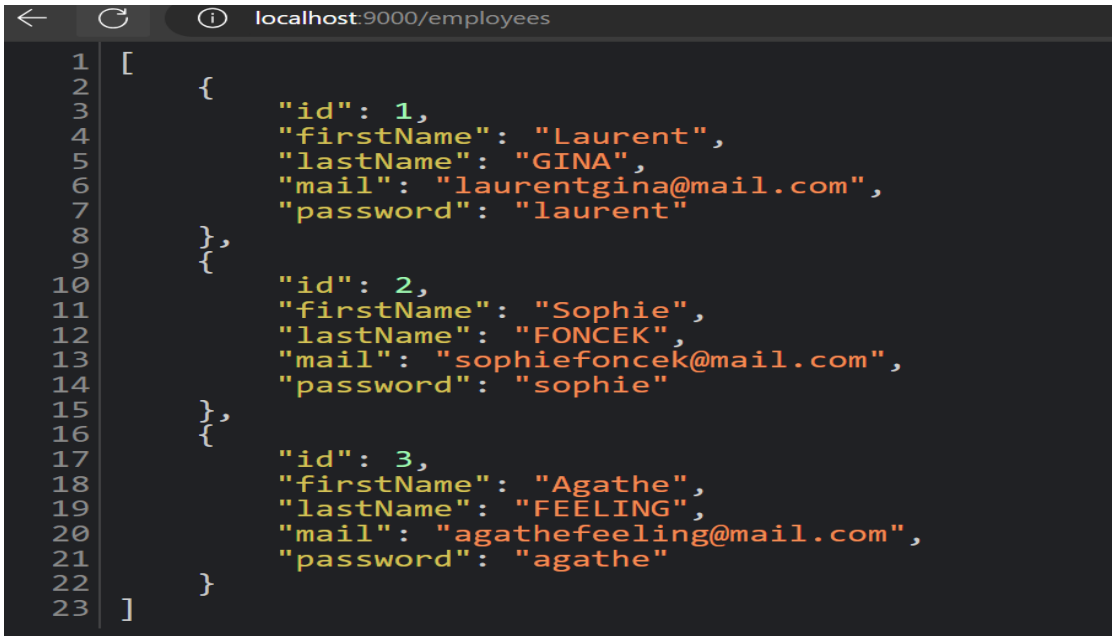
8.5 Exécution du microservice




```
: Starting ApiSbApplication using Java 1.8.0_92 on L31422978 with PID 14308
: No active profile set, falling back to 1 default profile: "default"
: Tomcat initialized with port(s): 9000 (http)
: H2 console available at '/h2-console'. Database available at 'jdbc:h2:mem:ce51ad87-6ee5-4e69-9bac-2c
: Tomcat started on port(s): 9000 (http) with context path ''
: Started ApiSbApplication in 7.731 seconds (JVM running for 8.522)
```

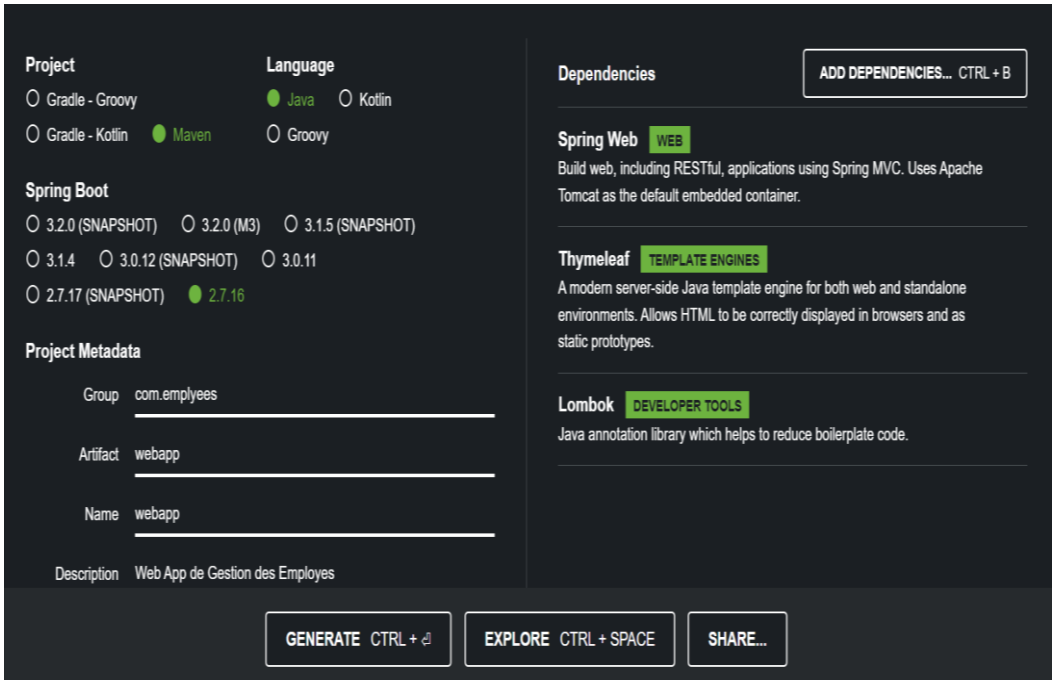
- Remarquer le n° de port de Tomcat
- Remarquer l'activation de la console de la base de données H2, avec la clé :

  `jdbc:h2:mem:ce51ad87-6ee5-4e69-9bac-2dc1383503e4`
- http://localhost:9000/h2-console/
- localhost:9000/employees

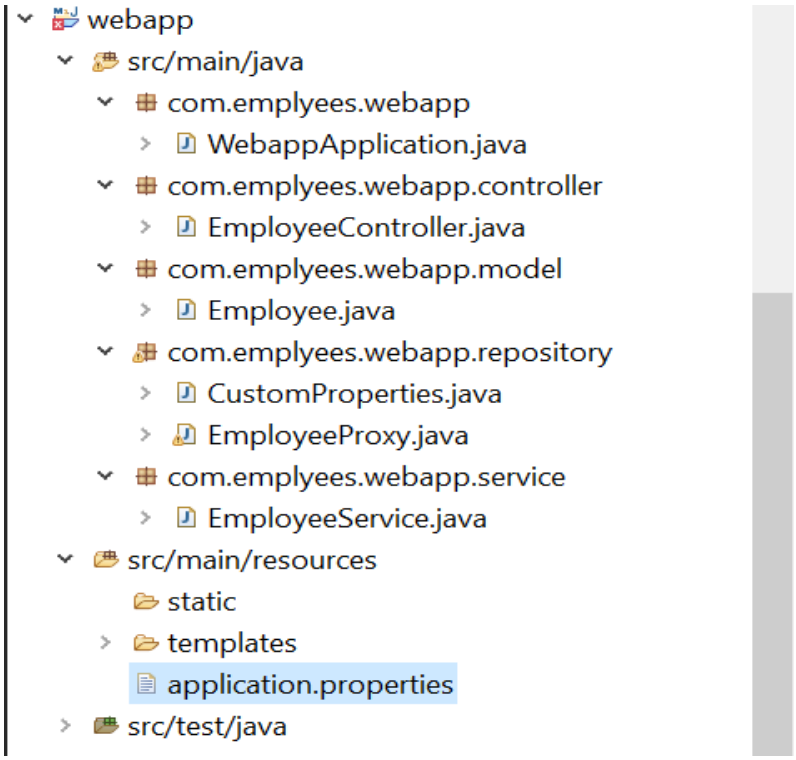

localhost:9000/employees
[
{
"id": 1,
"firstName": "Laurent",
"lastName": "GINA",
"mail": "laurentgina@mail.com",
"password": "laurent"
},
{
"id": 2,
"firstName": "Sophie",
"lastName": "FONCEK",
"mail": "sophiefoncek@mail.com",
"password": "sophie"
},
{
"id": 3,
"firstName": "Agathe",
"lastName": "FEELING",
"mail": "agathefeeling@mail.com",
"password": "agathe"
}
]

9. Partie(2) : Développement de la WebApp

9.1 https://start.spring.io/



- En plus de Spring Web, le starter Thymeleaf est l'un des moteurs de template (template engine) les plus couramment utilisés.
- Un moteur de template HTML va nous aider à formater la page HTML que nous voulons renvoyer.



9.2 Configuration personnalisée de la WebApp

➔ application.properties

```
spring.application.name=webapp
server.port=9001
logging.level.root=ERROR
logging.level.com.emplyees=INFO
logging.level.org.springframework.boot.web.embedded.tomcat=INFO
com.emplyees.webapp.apiUrl=http://localhost:9000
```

➔ Comment lire la propriété personnalisée « com.emplyees.webapp.apiUrl » ?

Solution → Créez le bean associé : créer une nouvelle classe nommée CustomProperties :

```
package com.emplyees.webapp.repository;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Configuration;
import lombok.Data;

@Configuration
@ConfigurationProperties(prefix = "com.emplyees.webapp")
@Data
public class CustomProperties {
   private String apiUrl;
}
```

On peut créer des propriétés et les manipuler dans le code, notamment grâce à l'annotation @ConfigurationProperties

@Configuration : permet de déclarer la classe en tant que bean de configuration.

@ConfigurationProperties(prefix = "com.emplyees.webapp") : demande à Spring de charger les properties qui commencent par "com.emplyees.webapp" au sein des attributs de la classe.

Afin de tester le bon chargement de la propriété personnalisée apiUrl :

```
package com.emplyees.webapp;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import com.emplyees.webapp.repository.CustomProperties;
import lombok.Data;

@Data
@SpringBootApplication
public class WebappApplication implements CommandLineRunner {
   @Autowired
   private CustomProperties properties;

   public static void main(String[] args) {
```

```
        SpringApplication.run(WebappApplication.class, args);
    }
    @Override
    public void run(String... args) throws Exception {
                System.out.println("CustomProperties properties.getApiUrl():  "+
properties.getApiUrl());
    }
}
```

➔ Exécution :

```
[main] com.emplyees.webapp.WebappApplication    :

 : Tomcat initialized with port(s): 9001 (http)

 : Tomcat started on port(s): 9001 (http) with context path ''

 : Started WebappApplication in 2.941 seconds (JVM running for 3.353)

CustomProperties properties.getApiUrl():  http://localhost:9000
```

➔ En résumé :

- La structure des packages reste le standard : controller/service/repository/model.
- Le fichier application.properties est la source de propriétés.
- On peut créer des propriétés et les manipuler dans mon code, notamment grâce à l'annotation @ConfigurationProperties.

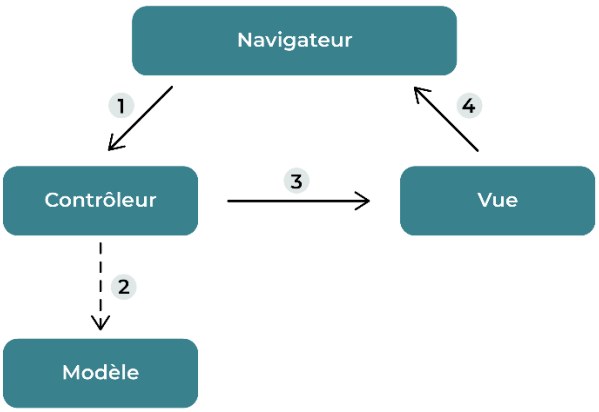
9.3 Développement de la WebApp

➔ L'objectif du ce projet est :

- Communiquer avec le microservice EmployeesMngt pour récupérer ou modifier les données des employés via l'API REST.
- Appliquer des traitements métiers spécifiques à l'application web.
- Afficher les pages web permettant de lister les employés, créer un nouvel employé, modifier un employé existant et supprimer un employé existant.

➔ Une approche MVC sera mise en œuvre :

MVC




```
package com.emplyees.webapp.model;
import lombok.Data;
@Data
public class Employee {
   private Integer id;
   private String firstName;
   private String lastName;
   private String mail;
   private String password;
}
```

9.4 Implémentation la communication entre l'application web et l'API REST

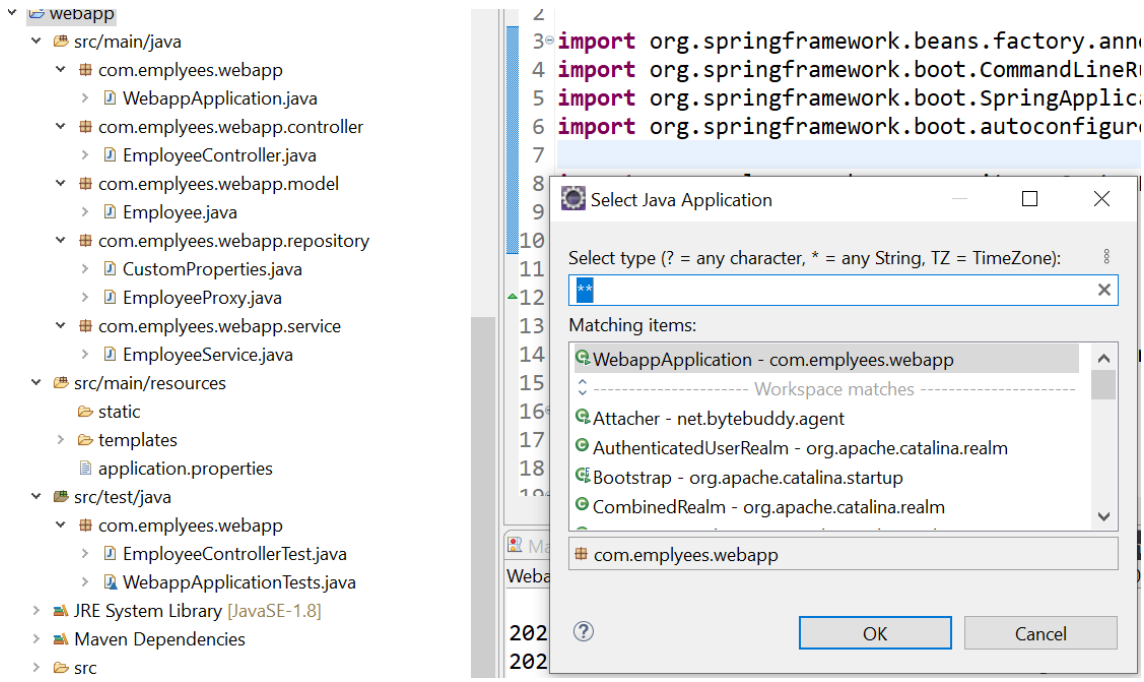
- Le starter Spring Web fournit le code nécessaire pour cela. Nous allons nous servir de la classe RestTemplate.
- RestTemplate permet d'exécuter une requête HTTP: On a besoin de fournir l'URL, le type de requête (GET, POST, etc.), et le type d'objet qui sera retourné.

```
package com.emplyees.webapp.repository;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.ParameterizedTypeReference;
import org.springframework.http.HttpMethod;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Component;
import org.springframework.web.client.RestTemplate;

import com.emplyees.webapp.model.Employee;
import lombok.extern.slf4j.Slf4j;
```

```java
@Slf4j
@Component
public class EmployeeProxy {

    @Autowired
    private CustomProperties props;

    /**
    * Get all employees
    * @return An iterable of all employees
    */

    public Iterable<Employee> getEmployees() {
        String baseApiUrl = props.getApiUrl();
        String getEmployeesUrl = baseApiUrl + "/employees";

        System.out.println("EmployeeProxy *** getEmployees()
getEmployeesUrl  " +getEmployeesUrl);


        RestTemplate restTemplate = new RestTemplate();
        ResponseEntity<Iterable<Employee>> response =
restTemplate.exchange(
                getEmployeesUrl,
                HttpMethod.GET,
                null,
                new
ParameterizedTypeReference<Iterable<Employee>>() {}
                );
        System.out.println("EmployeeProxy *** getEmployees()
response.getBody()  " +response.getBody());

        log.debug("Get Employees call " +
response.getStatusCode().toString());

        return response.getBody();
    }

}
```

- RestTemplate grâce à la methode exchange convertit le résultat JSON en objet Java
- le type retour, peut être un objet ParameterizedTypeReference car /employees renvoie un objet Iterable<Employee>.
- Sil'endpoint renvoie un objet simple, alors il suffira d'indiquer <Object>.class.
- On récupère notre objet Iterable<Employee> grâce à la méthode getBody() de l'objet Response.

➔ Test de la WebApp :



- Ne pas oublier de démarrer le microservice EmployeesMngt !
- Tester unitairement et vérifier sur la console :

```
EmployeeProxy *** getEmployees() getEmployeesUrl  http://localhost:9000/employees
EmployeeProxy *** getEmployees() response.getBody()  [Employee(id=1, firstName=Laurent,
lastName=GINA, mail=laurentgina@mail.com, password=laurent), Employee(id=2,
firstName=Sophie, lastName=FONCEK, mail=sophiefoncek@mail.com, password=sophie),
Employee(id=3, firstName=Agathe, lastName=FEELING, mail=agathefeeling@mail.com,
password=agathe)]
```

- Continuer le développement des autres couches :

```
package com.emplyees.webapp.service;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.emplyees.webapp.model.Employee;
import com.emplyees.webapp.repository.EmployeeProxy;

import lombok.Data;
@Data
@Service
public class EmployeeService {

   @Autowired
   private EmployeeProxy employeeProxy;

   public Employee getEmployee(final int id) {
      return employeeProxy.getEmployee(id);
```

```
    }

    public Iterable<Employee> getEmployees() {
        return employeeProxy.getEmployees();
    }

    public void deleteEmployee(final int id) {
        employeeProxy.deleteEmployee(id);;
    }

     public Employee saveEmployee(Employee employee) {
        Employee savedEmployee;

        // Règle de gestion : Le nom de famille doit être mis en majuscule.
        employee.setLastName(employee.getLastName().toUpperCase());

        if(employee.getId() == null) {
            // Si l'id est nul, alors c'est un nouvel employé.
            savedEmployee = employeeProxy.createEmployee(employee);
        } else {
            savedEmployee = employeeProxy.updateEmployee(employee);
        }
        return savedEmployee;
    }
}
```

- Implication de la couche Vue (Front end) :
- Lors du choix des starters, nous avons sélectionné **Spring Web**, et également le moteur de template Thymeleaf.
- Au niveau du répertoire Templates qui a pour vocation à contenir les fichiers HTML. Suivre les étapes à suivre :
    - Ajouter un fichier home.html qui servira de page d'accueil. Cela correspond à écrire le code 'Vue' de MVC.
    - Créer une classe nommée EmployeeController dans le package controller.
- La classe EmployeeController sera annotée afin qu'elle soit détectée comme un bean, en utilisant l'annotation @Controller. De nouveau c'est une spécialisation de @Component.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhml" xmlns:th="http://www.thymeleaf.org">
  <head>  <meta charset="UTF-8">
<title>Liste des employées : Modele MVC avec le Template Spring: Thymeleaf</title>
</head>
  <body>
```

```
    <h2 class="h2">Liste des employées</h2>
    <table><thead>
        <tr>
          <th>Prénom</th>
          <th>Nom</th>
        </tr>
      </thead>
      <tbody>
        <tr th:if="${employees.empty}">
          <td colspan="3">Aucun employée en base de données</td>
        </tr>
        <tr th:each="employee: ${employees}">
          <td><span th:text="${employee.firstName}"> Prénom </span></td>
          <td><span th:text="${employee.lastName}"> Nom </span></td>
        </tr>
      </tbody>
    </table>
    Abdel : This a test
  </body>
</html>
```

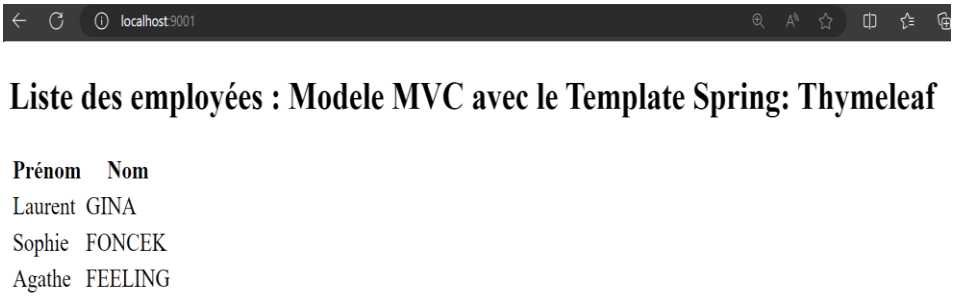
```
package com.emplyees.webapp.controller;

import org.springframework.web.bind.annotation.GetMapping;
import com.emplyees.webapp.model.Employee;
import com.emplyees.webapp.service.EmployeeService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;

@Controller
public class EmployeeController {

        @Autowired
        private EmployeeService service;

        @GetMapping("/")
        public String employees(Model model) {
                System.out.println("EmployeeController *** employees(Model)  ");

          Iterable<Employee> listEmployee = service.getEmployees();

          model.addAttribute("employees", listEmployee);

          return "home";
        }}
```

➔ Le concept est le suivant :

- L'annotation spécifie le type de requête HTTP et l'URL correspondante.
- Le texte "home" retourné correspond au nom du fichier HTML.

- À l'appel de l'URL racine de l'application web, la méthode home() sera automatiquement exécutée, et Spring renverra automatiquement une réponse HTTP contenant dans son corps (donc le body HTTP) le contenu du fichier home.html.
- L'objet Model (org.springframework.ui.Model) a été ajouté en paramètre de la méthode home(). Grâce à cela, Spring se charge de nous fournir une instance de cet objet.
- Puis, dans le corps de la méthode, on utilise la méthode addAttribute qui permet d'ajouter au Model un objet.
- Le premier paramètre spécifie le nom (de mon choix) et le deuxième l'objet (ici, la liste des employés en Iterable).

- `Tester unitairement : http://localhost:9001/`



- Les instructions **th:if** et **th:each** qui permettent respectivement d'implémenter une condition et une itération, fonctionnalités non présentes en HTML. Thymeleaf me fournit donc la capacité d'écrire **un code HTML dynamique**.
- La syntaxe **${nom de l'attribut}** permet d'accéder à un objet placé comme attribut dans le Model.
- Notons également que Thymeleaf comprend la programmation objet, et que la syntaxe **${$objet.attribut}** fonctionne.

9.5 *Récupération des données provenant d'un formulaire*

9.5.1 Situation (1) : la donnée est transmise par URL

- Exemple de suppression d'un employé par exemple
- La méthode deleteEmployee possède un paramètre nommé id de type int ;
- Le point clé est l'annotation @PathVariable qui signifie que ce paramètre est présent dans l'URL de requête http://localhost:9001/deleteEmployee/1
- Ajouter la méthode ci-après au Controller WebApp :

```
/**
 * Delete - Delete an employee
 * @param id - The id of the employee to delete
 */
@DeleteMapping("/employee/{id}")
public void deleteEmployee(@PathVariable("id") final Long id) {
        employeeService.deleteEmployee(id);
}
```

```
<td> <a th:href="@{/deleteEmployee/{id}(id=${employee.id})}">Supprimer</a></td>
```

- Coté IHM Thymeleaf : On utilise l'attribut th:href de Thymeleaf, et la syntaxe @{} permet de définir une URL.
- Le chemin /deleteEmployee/ est complété par l'id à fournir grâce à la syntaxe : {id}(id==${employee.id}).

Liste des employées : Modele MVC avec le Template Spring: Thymeleaf

Prénom	Nom	
Laurent	GINA	Supprimer
Agathe	FEELING	Supprimer

9.5.2 Situation (2) : la donnée est transmise par un formulaire

- La méthode saveEmployee est annotée @PostMapping et non @GetMapping.
- il s'agit de traiter la validation d'un formulaire.
- L'autre point clé est le paramètre de la méthode "@ModelAttribute Employee employee".
- @ModelAttribute est l'annotation clé : Cette annotation permet à Spring de récupérer les données saisies dans les champs du formulaire et de construire un objet Employee correspondant.
- Ajouter la méthode ci-après au Controller WebApp :

```
@PostMapping("/saveEmployee")
public ModelAndView saveEmployee(@ModelAttribute Employee employee) {
  service.saveEmployee(employee);
  return new ModelAndView("redirect:/");
}
```

Au niveau du fichier home.html

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhml"
	xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="UTF-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-scale=1">

<link rel="stylesheet"
	href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
	integrity="sha384-
BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u"
	crossorigin="anonymous">

<link rel="stylesheet"
	href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap-theme.min.css"
	integrity="sha384-
rHyoN1iRsVXV4nD0JutlnGaslCJuC7uwjduW9SVrLvRYooPp2bWYgmgJQIXwl/Sp"
	crossorigin="anonymous">

<title>Employee Web Application</title>

<style>
	body {
		padding-top: 50px;
	}
	.special {
		padding-top:50px;
	}
</style>
</head>
<body>

	<nav class="navbar navbar-inverse navbar-fixed-top">
		<div class="container">
			<div class="navbar-header">
				<a class="navbar-brand" href=".">Système de gestion des
employées</a>
			</div>
		</div>
	</nav>


	<div class="container special">
		<h2 class="h2">Liste des employées</h2>
		<div class="table-responsive">
			<table class="table table-striped table-sm">
				<thead>
					<tr>
						<th>Prénom</th>
						<th>Nom</th>
						<th>Action</th>
```

```
                                    </tr>
                                </thead>
                                <tbody>
                                    <tr th:if="${employees.empty}">
                                        <td colspan="3">Aucun employée en base de
données</td>
                                    </tr>
                                    <tr th:each="employee: ${employees}">
                                        <td><span th:text="${employee.firstName}">
Prénom </span></td>
                                        <td><span th:text="${employee.lastName}">
Nom </span></td>
                                        <td>
                                            <a
th:href="@{/updateEmployee/{id}(id=${employee.id})}"><button class="btn btn-
info">Modifier</button></a>
                                            <a
th:href="@{/deleteEmployee/{id}(id=${employee.id})}"><button class="btn btn-
danger">Supprimer</button></a>
                                        </td>
                                    </tr>
                                </tbody>
                            </table>
                        </div>
                        <h4><a th:href="@{/createEmployee}">Ajouter un nouvel employée</a></h4>

                    </div>

                    <script

                    src="https://ajax.googleapis.com/ajax/libs/jquery/1.12.4/jquery.min.js"></script>
                    <script
                        src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"
                        integrity="sha384-
Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNIcPD7Txa"
                        crossorigin="anonymous"></script>
</body>
</html>
```

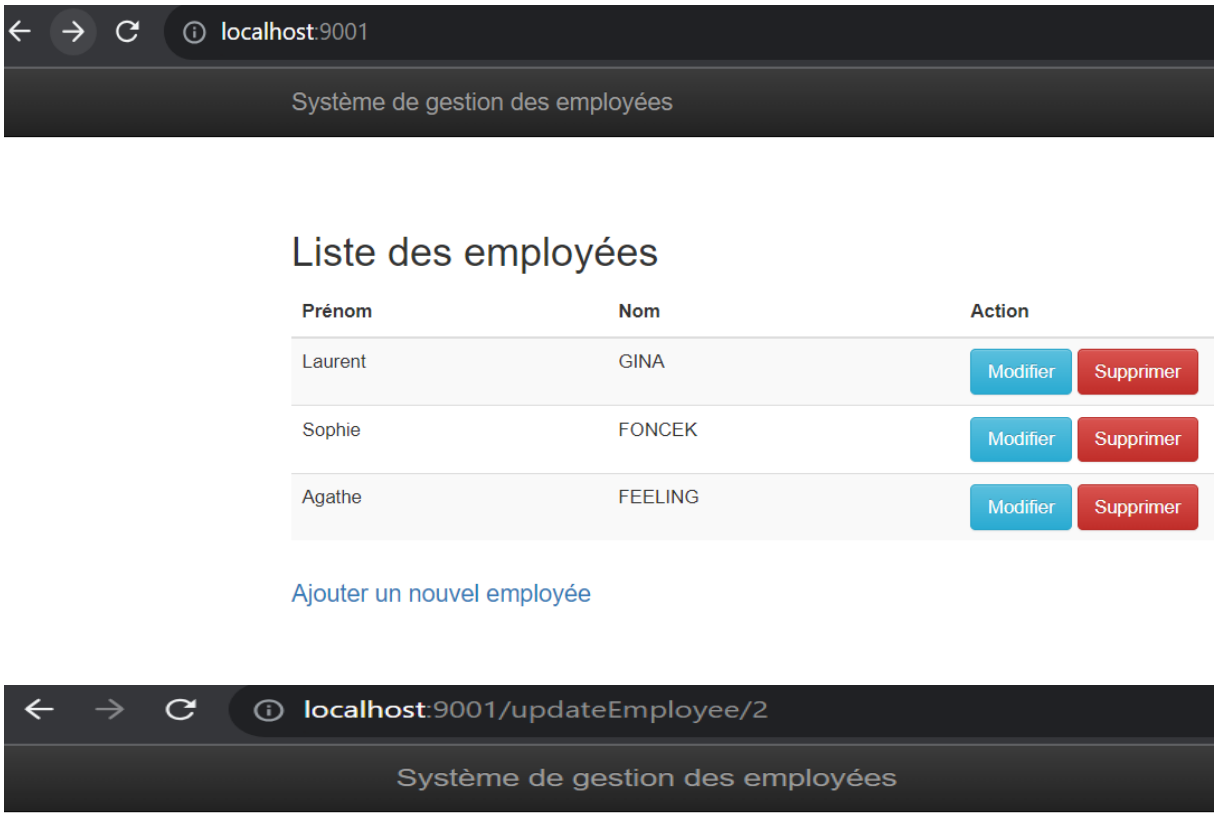
- Créer le fichier du formulaire formUpdateEmployee.html

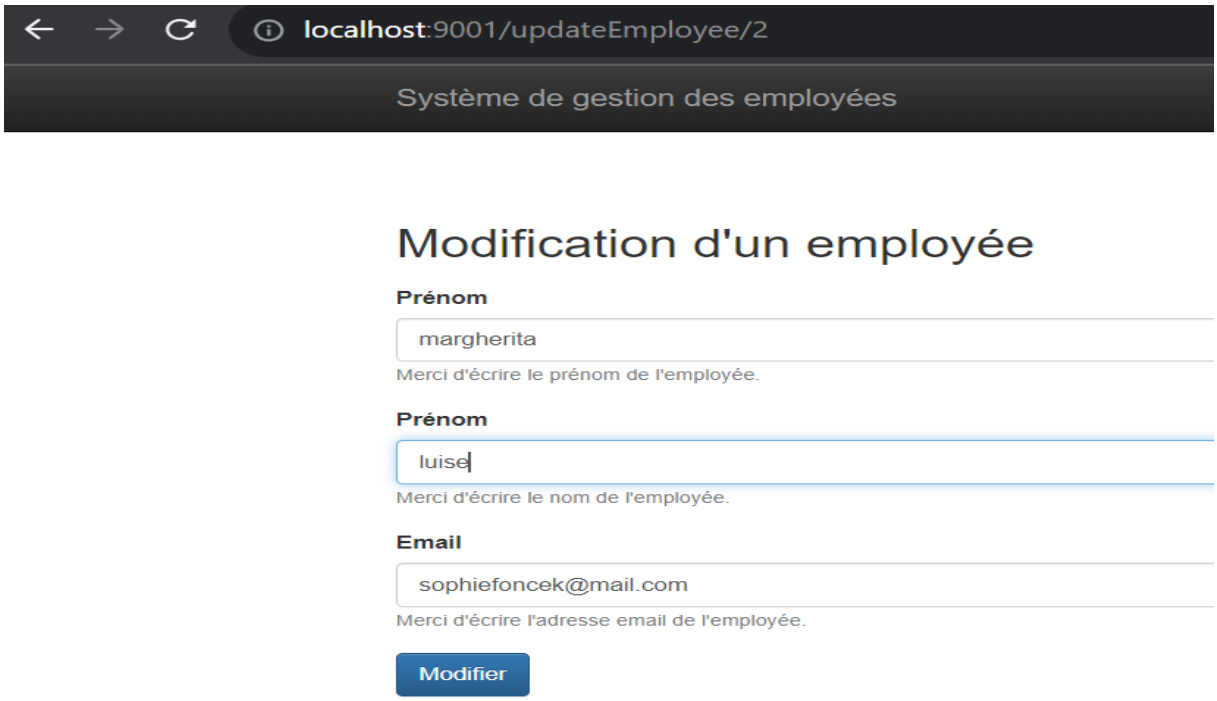
```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhml"
xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="UTF-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-
scale=1">
```

```
<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u"
crossorigin="anonymous">

<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap-theme.min.css"
integrity="sha384-rHyoN1iRsVXV4nD0JutlnGaslCJuC7uwjduW9SVrLvRYooPp2bWYgmgJQIXwl/Sp"
crossorigin="anonymous">

<title>Employee Web Application</title>

<style>
body {
padding-top: 50px;
}

.special {
padding-top: 50px;
}
</style>
</head>
<body>

<nav class="navbar navbar-inverse navbar-fixed-top">
<div class="container">
<div class="navbar-header">
<a class="navbar-brand" href=".">Système de gestion des employées</a>
</div>
</div>
</nav>


<div class="container special">

<h2 class="h2">Modification d'un employée</h2>
<div>
<form method="post"
th:action="@{/saveEmployee}" th:object="${employee}">
<div class="form-group">
<input
type="hidden" th:field="*{id}" class="form-control">
</div>
<div class="form-group">
```

```
<label for="firstNameInput">Prénom</label>
<input
type="text" th:field="*{firstName}" class="form-control"
id="firstNameInput"
aria-describedby="firstNameHelp" placeholder="Saisir le prénom">
<small
id="firstNameHelp" class="form-text text-muted">Merci d'écrire le
prénom de l'employée.</small>
</div>
<div class="form-group">
<label for="lastNameInput">Prénom</label>
<input
type="text" th:field="*{lastName}" class="form-control"
id=""lastNameInput""
aria-describedby="lastNameHelp" placeholder="Saisir le prénom">
<small
id="lastNameHelp" class="form-text text-muted">Merci d'écrire le
nom de l'employée.</small>
</div> <div class="form-group"><label
for="mailInput">Email</label>
<input type="text" th:field="*{mail}" class="form-control"
id="mailInput"
aria-describedby="mailHelp" placeholder="Saisir l'adresse email">
<small
id="mailHelp" class="form-text text-muted">Merci d'écrire
l'adresse email de l'employée.</small>
</div><button type="submit" class="btn btn-
primary">Modifier</button> </form></div> </div>
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/1.12.4/jquery.mi
n.js"></script>
<script
src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.
min.js"
integrity="sha384Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9m
GCD8wGNIcPD7Txa"crossorigin="anonymous"></script>
</body></html>
```

- **th: object=${employee}** fait le lien avec le ModelAttribute.
- **th:field** donne la correspondance avec les attributs de l'objet associé. Vous pouvez noter la syntaxe particulière : **"*{firstName}"**.


localhost:9001
Système de gestion des employées
Liste des employées
Prénom
Nom
Action
Laurent
GINA
Modifier
Supprimer
Sophie
FONCEK
Modifier
Supprimer
Agathe
FEELING
Modifier
Supprimer
Ajouter un nouvel employée


localhost:9001/updateEmployee/2
Système de gestion des employées
Modification d'un employée
Prénom
margherita
Merci d'écrire le prénom de l'employée.
Prénom
luise
Merci d'écrire le nom de l'employée.
Email
sophiefoncek@mail.com
Merci d'écrire l'adresse email de l'employée.
Modifier

Remarquer que le Nom de luise a été mis en majuscule conformément à la règle métier du Service de la WebApp.

➔ En résumé :

- Créer une application web avec Spring Boot correspond à suivre l'architecture MVC :
- Le modèle correspond aux classes Java qui représentent les données à manipuler ;
- La vue correspond aux fichiers HTML qui seront retournés à l'utilisateur
- Le contrôleur correspond aux classes Java annotées @Controleur qui font du mapping d'URL (avec par exemple @GetMapping) ;
- Les couches service et repository sont utilisées par les contrôleurs pour obtenir les données à fournir à la vue.
- RestTemplate est l'objet clé pour communiquer avec une API. Non seulement il exécute des requêtes HTTP, mais en plus il transforme le résultat JSON en objet Java.