Introduction à Spring Boot

Spring Boot est un framework Java né en 2014 basé sur le framework Spring, conçu pour simplifier le développement d'applications Java robustes et évolutives. Il permet de créer des applications indépendantes, prêtes pour la production, tout en réduisant le besoin de configuration manuelle. Il est apparu suite à la complexité jugée très (trop ?) importante des frameworks java originels qu'il remplace/complète (D'abord J2EE, puis Spring en 2002).

Partie 1 : découverte

Qu'est-ce que Spring Boot ?

Spring Boot est donc une extension du framework Spring qui permet de :

- **Faciliter la configuration** : Grâce à l'auto-configuration, Spring Boot élimine la complexité de la configuration manuelle.
- **Créer des applications autonomes** : Pas besoin de déployer sur un serveur d'application, vous pouvez directement exécuter l'application comme un simple programme Java.
- **Rendre les applications prêtes pour la production** avec des outils de surveillance, des systèmes de gestion et des configurations faciles à intégrer.

Avantages de Spring Boot :

- **Démarrage rapide** : Une application peut être rapidement créée et exécutée grâce à des dépendances préconfigurées.
- **Auto-configuration** : Spring Boot devine les configurations nécessaires en fonction des bibliothèques présentes dans le classpath.
- **Standalone** : Il crée une application packagée dans un JAR ou un WAR pouvant être exécuté directement.
- **Facile à intégrer avec des bases de données, des systèmes de messagerie, etc.**
- **Productivité accrue** : Réduction du code "boilerplate" et des tâches répétitives.

Ps : pour le code BoilerPlate, (passe-partout) voir la section suivante de wikipédia et mettre en application Lombok en Java par exemple pour illustrer le concept.

Structure d'une Application Spring Boot

Une application Spring Boot typique est structurée comme suit :

```
com.example.demo
|
├── src
|   ├── main
|   |   ├── java
|   |   |   └── com
|   |   |       └── example
|   |   |           └── demo
|   |   |               ├── DemoApplication.java
|   |   |               └── controllers
|   |   |                   └── HomeController.java
|   |   └── resources
|   |       ├── application.properties
|   |       └── static
|   |           └── index.html
```

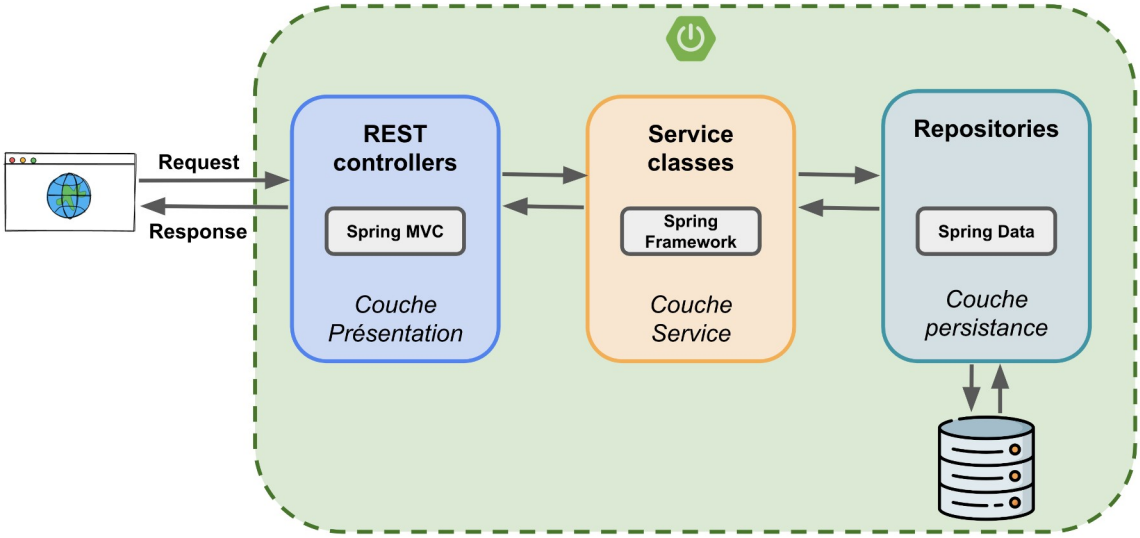
DemoApplication.java : Point d'entrée de l'application, généralement annoté avec @SpringBootApplication.

Controllers : Gère les requêtes HTTP. Les classes sont annotées avec @RestController ou @Controller.

application.properties ou **application.yml** : Fichier de configuration centralisé pour la personnalisation.

Architecture

Spring Boot n'impose aucune architecture pré établie. Néanmoins, il est courant de l'utiliser dans un cadre d'architecture classique en couches de type n-tiers.



Environnement

Spring boot nécessite essentiellement 4 outils :

1. Un environnement de développement Java
2. Un éditeur de code (aussi appelé IDE pour Integrated Development Environment) : IntelliJ IDEA Community Edition
3. Un outil de gestion de code source : Git (cet outil nous servira à récupérer des projets)
4. Un outil de gestion du cycle de vie de notre projet : Maven

Concernant la version de Java utilisée, nous allons installer la dernière version LTS en date.

LTS signifie Long Term Support, les versions LTS sont celles qui reçoivent des mises à jour régulières pendant une période donnée, généralement 2 à 3 ans, avant qu'une nouvelle version ne soit labellisée LTS. C'est le choix privilégié dans le monde de l'entreprise.

A noter que nous n'aurons pas besoin d'installer Maven, nous intègrerons dans le projet exemple un utilitaire, appelé Maven Wrapper, qui embarque directement l'outil. Cela évite de devoir l'installer et d'être sûr d'avoir la bonne version.

Création d'une application

Étape 1 : Créer un projet avec Spring Initializr

- Allez sur le site Spring Initializr.
- Choisissez les dépendances nécessaires (ex. Spring Web, Spring Data JPA, MySQL Driver).
- Cliquez sur "Generate" pour télécharger le projet ZIP, puis importez-le dans votre IDE.

Étape 2 : Écrire une application simple

```
// DemoApplication.java
package com.example.demo;

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

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

Étape 3 : Créer un Controller

```
// HomeController.java

package com.example.demo.controllers;

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

@RestController
public class HomeController {

   @GetMapping("/")
   public String home() {
      return "Hello, Spring Boot!";
   }
}
```

Étape 4 : Exécuter l'application

- Compilez et exécutez avec mvn spring-boot:run.
- Visitez http://localhost:8080 dans votre navigateur pour voir le message.

Félicitations, vous venez de créer votre première application Spring Boot.

A vous : tester la création du projet Hello World à votre tour sur votre poste.

Activité 1 : création d'une route d'API pour surveiller le système

Nous allons dans cette première activité réaliser une API Spring Boot qui servira à monitorer le système sur lequel elle est installée. Ceci permettra par la suite de consulter l'usage du processeur, l'occupation de la mémoire vive et la place restante sur les disques durs.

Premier contrôleur

Créez un nouveau projet et réalisons ce contrôleur.

Observez le code fourni dans le dossier activite1/metrics1

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

import java.io.File;
import java.lang.management.ManagementFactory;
import java.util.HashMap;
import java.util.Map;
import com.sun.management.OperatingSystemMXBean;

@RestController
public class SystemMetricsController {

   @GetMapping("/metrics")
   public Map<String, Object> getSystemMetrics() {
      // Récupérer les informations sur le système
      OperatingSystemMXBean osBean = (OperatingSystemMXBean)
ManagementFactory.getOperatingSystemMXBean();

      // Récupérer l'utilisation du CPU
      double cpuLoad = osBean.getSystemCpuLoad() * 100;

      // Récupérer l'utilisation de la mémoire
      long totalMemory = osBean.getTotalPhysicalMemorySize();
      long freeMemory = osBean.getFreePhysicalMemorySize();
      long usedMemory = totalMemory - freeMemory;

      // Convertir la mémoire de bytes à mégaoctets (MO)
      long totalMemoryMB = totalMemory / (1024 * 1024);
      long usedMemoryMB = usedMemory / (1024 * 1024);
      long freeMemoryMB = freeMemory / (1024 * 1024);
```

```
        // Récupérer l'utilisation des disques durs
        File[] roots = File.listRoots();
        Map<String, Object> diskMetrics = new HashMap<>();
        for (File root : roots) {
            long totalDiskSpace = root.getTotalSpace() / (1024 * 1024); // en MO
            long freeDiskSpace = root.getFreeSpace() / (1024 * 1024); // en MO
            long usedDiskSpace = totalDiskSpace - freeDiskSpace;
            double diskUsagePercentage = (double) usedDiskSpace / totalDiskSpace * 100;

            // Ajouter les métriques du disque dans la map
            Map<String, Object> diskInfo = new HashMap<>();
            diskInfo.put("totalDiskSpaceMB", totalDiskSpace);
            diskInfo.put("usedDiskSpaceMB", usedDiskSpace);
            diskInfo.put("freeDiskSpaceMB", freeDiskSpace);
            diskInfo.put("diskUsagePercentage", diskUsagePercentage);

            diskMetrics.put(root.getAbsolutePath(), diskInfo);
        }

        // Stocker les résultats dans une Map
        Map<String, Object> metrics = new HashMap<>();
        metrics.put("cpuUsage", cpuLoad);
        metrics.put("totalMemoryMB", totalMemoryMB);
        metrics.put("usedMemoryMB", usedMemoryMB);
        metrics.put("freeMemoryMB", freeMemoryMB);
        metrics.put("diskMetrics", diskMetrics);

        return metrics;
    }
}
```

A vous :

Créez un nouveau contrôleur dans votre projet avec le code ci-dessus.

a. Quel est la route à appeler par l'utilisateur de l'API et avec quelle méthode http ?

b. Testez cette route dans votre navigateur et dans postman.

Ajout d'une base de données

A présent, nous allons faire en sorte d'enregistrer ces informations dans une base de données Mysql. Pour cela, nous allons ajouter la dépendance vers mysql dans le fichier pom.xml de Maven.

```
<dependency>

   <groupId>mysql</groupId>
   <artifactId>mysql-connector-java</artifactId>
   <version>8.0.33</version>
</dependency>
```

Puis nous créons 2 nouvelles tables dans notre base de données.

```
CREATE TABLE system_metrics_cpu_ram (
   id INT AUTO_INCREMENT PRIMARY KEY,
   cpu_usage DOUBLE,
   total_memory_mb BIGINT,
   used_memory_mb BIGINT,
   free_memory_mb BIGINT,
   created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE system_metrics_disk (
   id INT AUTO_INCREMENT PRIMARY KEY,
   disk_path VARCHAR(255),
   total_disk_space_mb BIGINT,
   used_disk_space_mb BIGINT,
   free_disk_space_mb BIGINT,
   disk_usage_percentage DOUBLE,
   cpu_ram_id INT,
   created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
   FOREIGN KEY (cpu_ram_id) REFERENCES system_metrics_cpu_ram(id)
);
```

Le code devient alors celui-ci (disponible dans activite1/metrics2)

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

import java.io.File;
import java.lang.management.ManagementFactory;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.HashMap;
```

```
import java.util.Map;
import com.sun.management.OperatingSystemMXBean;

@RestController
public class SystemMetricsController {

   // Injecter les propriétés du fichier application.properties
   @Value("${spring.datasource.url}")
   private String databaseUrl;

   @Value("${spring.datasource.username}")
   private String databaseUsername;

   @Value("${spring.datasource.password}")
   private String databasePassword;

   @GetMapping("/metrics")
   public Map<String, Object> getSystemMetrics() {
      // Récupérer les informations sur le système
      OperatingSystemMXBean osBean = (OperatingSystemMXBean)
ManagementFactory.getOperatingSystemMXBean();

      // Récupérer l'utilisation du CPU
      double cpuLoad = osBean.getSystemCpuLoad() * 100;

      // Récupérer l'utilisation de la mémoire
      long totalMemory = osBean.getTotalPhysicalMemorySize();
      long freeMemory = osBean.getFreePhysicalMemorySize();
      long usedMemory = totalMemory - freeMemory;

      // Convertir la mémoire de bytes à mégaoctets (MO)
      long totalMemoryMB = totalMemory / (1024 * 1024);
      long usedMemoryMB = usedMemory / (1024 * 1024);
      long freeMemoryMB = freeMemory / (1024 * 1024);

      // Insérer les nouvelles données CPU/RAM dans la base de données
      int cpuRamId = insertCpuRamMetrics(cpuLoad, totalMemoryMB, usedMemoryMB,
freeMemoryMB);

      // Récupérer l'utilisation des disques durs
      File[] roots = File.listRoots();
      Map<String, Object> diskMetrics = new HashMap<>();
      for (File root : roots) {
         long totalDiskSpace = root.getTotalSpace() / (1024 * 1024); // en MO
         long freeDiskSpace = root.getFreeSpace() / (1024 * 1024); // en MO
         long usedDiskSpace = totalDiskSpace - freeDiskSpace;
         double diskUsagePercentage = (double) usedDiskSpace / totalDiskSpace * 100;

         // Ajouter les métriques du disque dans la map
         Map<String, Object> diskInfo = new HashMap<>();
         diskInfo.put("totalDiskSpaceMB", totalDiskSpace);
```

```
            diskInfo.put("usedDiskSpaceMB", usedDiskSpace);
            diskInfo.put("freeDiskSpaceMB", freeDiskSpace);
            diskInfo.put("diskUsagePercentage", diskUsagePercentage);

            diskMetrics.put(root.getAbsolutePath(), diskInfo);

            // Insérer les données disque avec une référence vers cpuRamId
            insertDiskMetrics(cpuRamId, root.getAbsolutePath(), totalDiskSpace, usedDiskSpace,
freeDiskSpace, diskUsagePercentage);
        }

        // Stocker les résultats dans une Map
        Map<String, Object> metrics = new HashMap<>();
        metrics.put("cpuUsage", cpuLoad);
        metrics.put("totalMemoryMB", totalMemoryMB);
        metrics.put("usedMemoryMB", usedMemoryMB);
        metrics.put("freeMemoryMB", freeMemoryMB);
        metrics.put("diskMetrics", diskMetrics);

        return metrics;
    }

    // Méthode pour insérer les données CPU/RAM
    private int insertCpuRamMetrics(double cpuUsage, long totalMemoryMB, long usedMemoryMB,
long freeMemoryMB) {
        String query = "INSERT INTO system_metrics_cpu_ram (cpu_usage, total_memory_mb,
used_memory_mb, free_memory_mb) "
                + "VALUES (?, ?, ?, ?)";
        int id = -1;

        try (Connection conn = DriverManager.getConnection(databaseUrl, databaseUsername,
databasePassword);
             PreparedStatement pstmt = conn.prepareStatement(query,
PreparedStatement.RETURN_GENERATED_KEYS)) {

            pstmt.setDouble(1, cpuUsage);
            pstmt.setLong(2, totalMemoryMB);
            pstmt.setLong(3, usedMemoryMB);
            pstmt.setLong(4, freeMemoryMB);
            pstmt.executeUpdate();

            // Récupérer l'ID généré
            try (var rs = pstmt.getGeneratedKeys()) {
                if (rs.next()) {
                    id = rs.getInt(1);
                }
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
```

```
        return id;
    }

    // Méthode pour insérer les données disque
    private void insertDiskMetrics(int cpuRamId, String diskPath, long totalDiskSpaceMB, long
usedDiskSpaceMB,
                                   long freeDiskSpaceMB, double diskUsagePercentage) {
        String query = "INSERT INTO system_metrics_disk (disk_path, total_disk_space_mb,
used_disk_space_mb, "
                + "free_disk_space_mb, disk_usage_percentage, cpu_ram_id) "
                + "VALUES (?, ?, ?, ?, ?, ?)";

        try (Connection conn = DriverManager.getConnection(databaseUrl, databaseUsername,
databasePassword);
             PreparedStatement pstmt = conn.prepareStatement(query)) {

            pstmt.setString(1, diskPath);
            pstmt.setLong(2, totalDiskSpaceMB);
            pstmt.setLong(3, usedDiskSpaceMB);
            pstmt.setLong(4, freeDiskSpaceMB);
            pstmt.setDouble(5, diskUsagePercentage);
            pstmt.setInt(6, cpuRamId);

            pstmt.executeUpdate();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Cette fois, nous enregistrons les données du système dans une table séparée, les données sur les disques dans une autre. Les deux sont liés par une clé étrangère car on considère qu'un relevé réalise les 2 tâches à chaque appel de l'API.

Nous pouvons aussi remarquer la présence d'attributs :

```
@Value("${spring.datasource.url}")
private String databaseUrl;

@Value("${spring.datasource.username}")
private String databaseUsername;

@Value("${spring.datasource.password}")
private String databasePassword;
```

Il s'agit d'injections de valeurs issues du fichier de configuration application.properties. Vous touchez déjà du doigt ici le principe d'injection de dépendances présent dans beaucoup de framework et spring boot en particulier.

Il existe d'autres façon d'injecter notamment des objets issus d'une classe à part entière que nous verrons un peu plus tard.

Architecture en couches

Notre code actuel présente un défaut au regard des bonnes pratiques de conception d'une architecture classique pour une application web de type MVC. Il mélange contrôleur et modèle permettant l'accès aux données.

Il serait plus propre de séparer ces 2 aspects. C'est que nous allons faire ici en créant un Service qui sera chargé de s'occuper des accès à la base. On y délocalisera nos 2 méthodes d'enregistrements des données.

Ajoutez une nouvelle classe dans le dossier Service de votre projet et nommez le SystemMetricsService. Le code sera le suivant :

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

@Service
public class SystemMetricsService {

   // Injecter les propriétés du fichier application.properties
   @Value("${spring.datasource.url}")
   private String databaseUrl;

   @Value("${spring.datasource.username}")
   private String databaseUsername;

   @Value("${spring.datasource.password}")
   private String databasePassword;

   // Méthode pour insérer les données CPU/RAM
   public int insertCpuRamMetrics(double cpuUsage, long totalMemoryMB, long usedMemoryMB,
long freeMemoryMB) {
      String query = "INSERT INTO system_metrics_cpu_ram (cpu_usage, total_memory_mb,
used_memory_mb, free_memory_mb) "
            + "VALUES (?, ?, ?, ?)";
      int id = -1;

      try (Connection conn = DriverManager.getConnection(databaseUrl, databaseUsername,
databasePassword);
          PreparedStatement pstmt = conn.prepareStatement(query,
PreparedStatement.RETURN_GENERATED_KEYS)) {

         pstmt.setDouble(1, cpuUsage);
```

```
            pstmt.setLong(2, totalMemoryMB);
            pstmt.setLong(3, usedMemoryMB);
            pstmt.setLong(4, freeMemoryMB);
            pstmt.executeUpdate();

            // Récupérer l'ID généré
            try (var rs = pstmt.getGeneratedKeys()) {
                if (rs.next()) {
                    id = rs.getInt(1);
                }
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }

        return id;
    }

    // Méthode pour insérer les données disque
    public void insertDiskMetrics(int cpuRamId, String diskPath, long totalDiskSpaceMB, long
usedDiskSpaceMB,
                                  long freeDiskSpaceMB, double diskUsagePercentage) {
        String query = "INSERT INTO system_metrics_disk (disk_path, total_disk_space_mb,
used_disk_space_mb, "
                + "free_disk_space_mb, disk_usage_percentage, cpu_ram_id) "
                + "VALUES (?, ?, ?, ?, ?, ?)";

        try (Connection conn = DriverManager.getConnection(databaseUrl, databaseUsername,
databasePassword);
             PreparedStatement pstmt = conn.prepareStatement(query)) {

            pstmt.setString(1, diskPath);
            pstmt.setLong(2, totalDiskSpaceMB);
            pstmt.setLong(3, usedDiskSpaceMB);
            pstmt.setLong(4, freeDiskSpaceMB);
            pstmt.setDouble(5, diskUsagePercentage);
            pstmt.setInt(6, cpuRamId);

            pstmt.executeUpdate();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Pas de surprise, nous avons simplement le code que nous avions dans le contrôleur.

Ensuite, modifiez votre contrôleur pour obtenir ce code :

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

import java.io.File;
import java.lang.management.ManagementFactory;
import java.util.HashMap;
import java.util.Map;
import com.sun.management.OperatingSystemMXBean;
import sio.projet.camping.service.SystemMetricsService;

@RestController
public class SystemMetricsController {

   private final SystemMetricsService systemMetricsService;

   // Injecter le service dans le contrôleur
   public SystemMetricsController(SystemMetricsService systemMetricsService) {
      this.systemMetricsService = systemMetricsService;
   }

   @GetMapping("/metrics")
   public Map<String, Object> getSystemMetrics() {
      // Récupérer les informations sur le système
      OperatingSystemMXBean osBean = (OperatingSystemMXBean)
ManagementFactory.getOperatingSystemMXBean();

      // Récupérer l'utilisation du CPU
      double cpuLoad = osBean.getSystemCpuLoad() * 100;

      // Récupérer l'utilisation de la mémoire
      long totalMemory = osBean.getTotalPhysicalMemorySize();
      long freeMemory = osBean.getFreePhysicalMemorySize();
      long usedMemory = totalMemory - freeMemory;

      // Convertir la mémoire de bytes à mégaoctets (MO)
      long totalMemoryMB = totalMemory / (1024 * 1024);
      long usedMemoryMB = usedMemory / (1024 * 1024);
      long freeMemoryMB = freeMemory / (1024 * 1024);

      // Insérer les nouvelles données CPU/RAM dans la base de données
      int cpuRamId = systemMetricsService.insertCpuRamMetrics(cpuLoad, totalMemoryMB,
usedMemoryMB, freeMemoryMB);

      // Récupérer l'utilisation des disques durs
      File[] roots = File.listRoots();
      Map<String, Object> diskMetrics = new HashMap<>();
      for (File root : roots) {
         long totalDiskSpace = root.getTotalSpace() / (1024 * 1024); // en MO
         long freeDiskSpace = root.getFreeSpace() / (1024 * 1024); // en MO
```

```
        long usedDiskSpace = totalDiskSpace - freeDiskSpace;
        double diskUsagePercentage = (double) usedDiskSpace / totalDiskSpace * 100;

        // Ajouter les métriques du disque dans la map
        Map<String, Object> diskInfo = new HashMap<>();
        diskInfo.put("totalDiskSpaceMB", totalDiskSpace);
        diskInfo.put("usedDiskSpaceMB", usedDiskSpace);
        diskInfo.put("freeDiskSpaceMB", freeDiskSpace);
        diskInfo.put("diskUsagePercentage", diskUsagePercentage);

        diskMetrics.put(root.getAbsolutePath(), diskInfo);

        // Insérer les données disque dans la base de données
        systemMetricsService.insertDiskMetrics(cpuRamId, root.getAbsolutePath(), totalDiskSpace,
usedDiskSpace, freeDiskSpace, diskUsagePercentage);
    }

    // Stocker les résultats dans une Map
    Map<String, Object> metrics = new HashMap<>();
    metrics.put("cpuUsage", cpuLoad);
    metrics.put("totalMemoryMB", totalMemoryMB);
    metrics.put("usedMemoryMB", usedMemoryMB);
    metrics.put("freeMemoryMB", freeMemoryMB);
    metrics.put("diskMetrics", diskMetrics);

    return metrics;
  }
}
```

A vous :

a. Identifiez par quel moyen on donne les informations sur le service au contrôleur.

b. Pouvez vous trouver un endroit dans le code des autres fichiers ou on crée le contrôleur en passant le service en paramètre ?

Vous n'avez pas trouvé de réponse à la question b ? C'est bien normal.

Dans Spring Boot, l'appel au constructeur pour fournir l'instance du service injecté se fait automatiquement par le framework, grâce à son système d'inversion de contrôle (**Inversion of Control**, ou IoC) et d'injection de dépendances (**Dependency Injection**, ou DI).

Voici comment cela fonctionne en coulisses :

1. **Détection des composants avec les annotations** :
    - Spring scanne les classes de votre application à la recherche d'annotations telles que @RestController, @Service, @Component, etc. Lorsque vous démarrez votre application, Spring construit un **contexte d'application** (le conteneur IoC) qui maintient un registre des beans (objets gérés par Spring) créés et gérés automatiquement par le framework.

2. **Création des beans** :
    - Lorsqu'il rencontre une classe annotée avec @Service, comme dans le cas de votre classe SystemMetricsService, Spring instancie cette classe en tant que bean. Un bean est un objet Java géré par le framework qui l'instancie automatiquement et l'injecte dans le code là ou on en a besoin.
    - Par défaut, Spring crée une **instance unique** (singleton) de cette classe, ce qui signifie que l'instance est créée une seule fois et partagée dans toute l'application.
3. **Injection des dépendances dans le constructeur** :
    - Quand Spring détecte une classe annotée avec @RestController comme votre classe SystemMetricsController, il voit que cette classe a un constructeur qui prend un paramètre SystemMetricsService.
    - Grâce à son mécanisme d'injection de dépendances, Spring cherche dans son conteneur d'IoC un bean de type SystemMetricsService qui a déjà été créé.
    - Une fois trouvé, Spring appelle le constructeur de SystemMetricsController et passe ce bean (l'instance de SystemMetricsService) en tant qu'argument du constructeur.

Il est possible d'injecter la dépendance de notre service également à l'aide de l'annotation @Autowired. Voici ce que cela donne :

```
@RestController public class SystemMetricsController {
// Injection de dépendance automatique via @Autowired
@Autowired
private SystemMetricsService systemMetricsService;

// Suite du code du contrôleur
```

A vous :

a. Modifiez le contrôleur pour ajouter une route permettant de récupérer l'ensemble des relevés systèmes effectués sur le serveur.

b. Modifiez le contrôleur pour ajouter une route permettant de récupérer l'ensemble des relevés sur les disques effectués sur le serveur.

c. Ajoutez 2 routes qui permettront de supprimer un relevé système ou disque dont l'id est passé en paramètre. Si c'est un relevé système, il faudra aussi supprimer l'ensemble des relevés disques qui y sont associés.

Documentation de l'API

Il est possible de générer une interface de documentation qui vous permettra de tester directement vos routes d'API dans le navigateur. Pour cela, vous installerez le dépendance MAVEN suivante :

```
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
  <version>2.2.0</version>
</dependency>
```

Open-api est une spécification qui permet de documenter une API à l'aide de fichiers JSON, YAML ou d'annotations directement insérées dans les langages. L'interface Swagger est basée sur open-api et avec elle on va pouvoir générer une documentation des API facilement par lecture des fichiers JSON, YAML.

Après avoir ajouté la dépendance, rendez-vous dans votre navigateur sur l'adresse :

http://localhost:8080/swagger-ui/index.html

Faites quelques tests.

Pour plus de détails, consultez : https://bell-sw.com/blog/documenting-rest-api-with-swagger-in-spring-boot-3/

Partie 2 : gestion des configurations

Le travail sur Spring Boot est l'occasion d'adopter une façon de travailler professionnelle. Ainsi, dans le monde de l'entreprise, les développeurs doivent composer avec plusieurs configurations. La vision la plus simple dans un premier temps est de séparer son environnement de développement et son environnement de production. Nous allons faire cela avec notre base de données dans un premier temps. Pour nos tests et la phase de développement, nous aurons un serveur BDD de test et nous travaillerons sur les données de production lorsque nous activerons la configuration de production.

Voyons comment mettre cela en place…

Création de fichiers de configuration

Créez deux fichiers de propriétés dans le répertoire src/main/resources :

- **application-dev.properties** : Pour la configuration de développement.
- **application-prod.properties** : Pour la configuration de production.

Exemple de `application-dev.properties`

```
# Configuration de développement
spring.datasource.url=jdbc:mysql://localhost:3306/nom_de_votre_base_dev
spring.datasource.username=dev_user
spring.datasource.password=dev_password
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```

Exemple de `application-prod.properties`

```
# Configuration de production
spring.datasource.url=jdbc:mysql://production-server:3306/
nom_de_votre_base_prod
spring.datasource.username=prod_user
spring.datasource.password=prod_password
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```

Nous avons 2 fichiers différents qui vont correspondre à chacun des environnements.

A vous :

a. Préparez 2 fichiers sur votre poste, l'un ira sur une BDD de test locale tournant sous Wamp par exemple, l'autre ira sur un conteneur docker sur lequel on mettre la base de production.

b. Dans le fichier initial application.properties, contentez vous de laisser cette ligne :

spring.profiles.active=dev

c. Lancez le programme et constatez que vous êtes sur la bonne base.

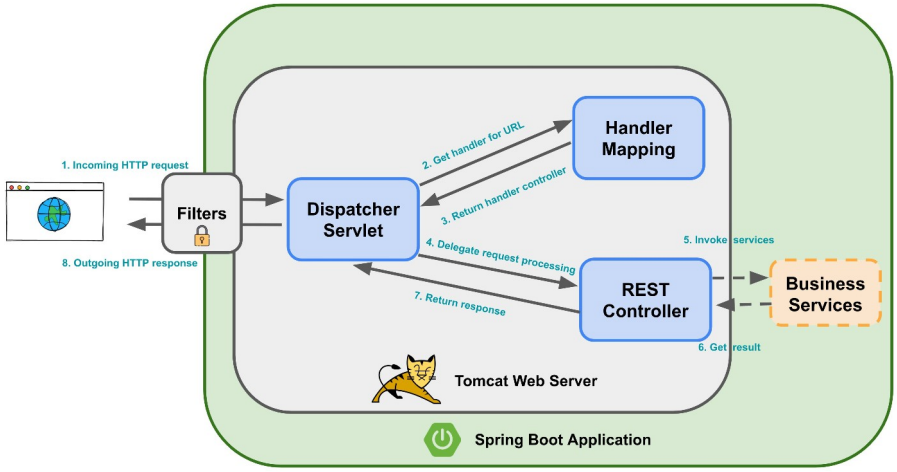
Nous pourrons aller plus loin dans la gestion des configurations un petit peu plus tard avec l'aide de Docker et l'automatisation du déploiement.

Partie 3 : sécurisation de l'application

Spring Security est le standard le plus répandu lorsqu'il s'agit de sécuriser une application Spring.
Le framework fournit des mécanismes d'autorisation et d'authentification ainsi que des protections contre les attaques malveillantes les plus répandues.

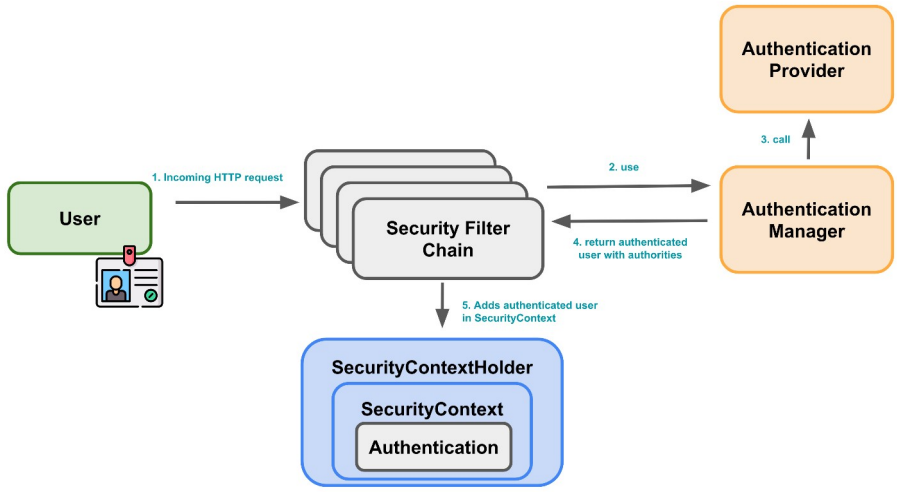
Spring Security est utilisé pour sécuriser des applications, notamment l'interaction entre un utilisateur et une application, ou entre 2 applications qui communiquent. En revanche, Spring Security ne permet pas de sécuriser des infrastructures.

Spring Security fonctionne via des mécanismes de filtres.



Spring Security se concentre sur l'authentification et l'autorisation.

- Authentification : reconnaître l'utilisateur et s'assurer que ce soit bien lui à l'origine de la demande de ressource
- Autorisation : une fois authentifié, s'assurer que l'utilisateur possède les bons droits.



Les principaux concepts et fonctionnalités

Filter chain

La plupart des fonctionnalités de Spring Security, lorsqu'elles sont activées, reposent sur un mécanisme de filters. Le framework met en oeuvre une chaîne de filtres bien définie dans un ordre logique. Par exemple, le filtre qui met en oeuvre l'authentification se déclenche avant celui des autorisations, en effet, il faut connaître l'utilisateur avant de vérifier ce qu'il a le droit de faire.

Security Context

Le Security Context est un élément central dans Spring Security.
Lorsque c'est nécessaire, le Security Context est consulté pour vérifier la présence d'un objet de type Authentication, c'est la présence de cet objet qui détermine si l'utilisateur est authentifié ou non. C'est donc lui qui stocke l'authentification en cours de validité.
Une fois cette étape franchie, l'utilisateur peut accéder aux ressources auxquelles il a le droit d'accéder selon ses autorisations.
S'il se déconnecte ou si sa session arrive à expiration, le Security Context est nettoyé et l'objet Authentication est supprimé. Cela évite notamment qu'une autre personne puisse utiliser ce compte.

Authentification

Spring Security offre aux développeurs différents mécanismes d'authentification : HTTP Basic Authentication, formulaire avec login/mot de passe, solution basée sur le protocole OAuth2, intégration avec le protocole LDAP mais aussi une intégration avec des fournisseurs d'identité externes.

Autorisations

Le framework fournit des fonctionnalités d'autorisation permettant un contrôle d'accès très fin basé sur des rôles, des permissions, des expressions ou encore de la logique personnalisée (des conditions que l'on peut coder). Avec ceci, les développeurs peuvent sécuriser des URLs, des méthodes ou même des objets.

Filtres de sécurité

Spring Security fonctionne à travers un enchaînement de filtres qui interceptent les requêtes entrantes pour déclencher l'authentification, vérifier les autorisations et les contraintes de sécurité. Les développeurs peuvent personnaliser cet enchaînement pour les besoins spécifiques de leur application.

Entêtes de sécurité

Spring Security permet aux développeurs de mettre en place des entêtes de sécurité dans les réponses HTTP des requêtes pour renforcer la protection contre les attaques courantes comme Clickjacking, Cross-Site Scripting (XSS) ou encore Man-in-the-Middle.

Gestion de la session

Spring Security offre des fonctionnalités relatives à la session utilisateur : gestion de l'expiration, contrôle des sessions concurrentes, protection contre les attaques de type session fixation. Les développeurs peuvent configurer la politique de gestion de session qui convient le mieux à leur application.

Intégration à Spring Boot

Spring Security s'intègre de manière transparente avec Spring Boot, notamment grâce à une dépendance clé en main qui lorsqu'elle est installée, va déclencher l'auto-configuration pour activer une sécurisation par défaut. Il est possible de customiser cette configuration par défaut via le mécanisme de properties ou de Java configuration.

Mise en place de la tables des données utilisateurs

Nous allons ajouter une nouvelle table dans notre base qui sera chargée de contenir les utilisateurs de notre application. Sa structure sera la suivante :

```
CREATE TABLE user (
                id INT AUTO_INCREMENT PRIMARY KEY,
                username VARCHAR(50) NOT NULL UNIQUE,
                email VARCHAR(100) NOT NULL UNIQUE,
                password VARCHAR(60) NOT NULL,  -- stocke le mot de passe haché
                role VARCHAR(20) NOT NULL,      -- rôle de l'utilisateur (ex: ADMIN, USER)
                created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
                updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP
);

INSERT INTO user (username, email, password, role) VALUES
                                ('admin_user', 'admin@example.com', '
$10$L.wiTqFpVOcP9HxvKSCiW.JHtd/VJTA/glHqYBYX7DOCGhxLiUP3e', 'ADMIN'),
                                ('basic_user', 'user@example.com', '
$10$9DFbDZEZBVFSs4g6OJ6c8e6t9iGHS5Di7qTxGOuW4lY6JWPPdAumm', 'USER');

-- password123 et userpassword comme mots de passe en bcrypt
```

Nous voyons que le rôle ici est simplement représenté par un champ dans la table, on sera soit administrateur, soit utilisateur basique. On aurait pu faire plus compliqué en attribuant plusieurs rôles à un même utilisateur en créant une table user_roles mais restons simples pour le moment.

Création des classes de Modèle et du Service utilisateur avec l'ORM

Cette étape est l'occasion de découvrir l'ORM le plus répandu en Java : Hibernate à l'aide de l'interface JPA.

JPA est l'interface permettant de faciliter le travail en Java avec les bases de données représentées par des objets. C'est ce qu'on appelle un outil ORM (Object Relationnal Mapping). Hibernate se base sur JPA pour fournir une implémentation de cette interface de programmation.

Nous allons commencer par créer une classe entité représentant un utilisateur.

A vous : dans un sous dossier entity, ajoutez la classe User dont le corps sera le suivant :

```
import jakarta.persistence.*;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;
import java.util.Collection;
import java.util.Collections;
import java.util.List;
import java.util.stream.Collectors;

@Entity
public class User implements UserDetails {

   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
   private String username;
   private String password;

   @Enumerated(EnumType.STRING)
   private Role role;  // Le rôle unique

   @Override
   public Collection<? extends GrantedAuthority> getAuthorities() {
      // Retourne le rôle unique de l'utilisateur sous forme de collection avec un seul élément
      return Collections.singletonList(new SimpleGrantedAuthority("ROLE_"+this.role.toString()));
   }
   @Override
   public String getPassword() {
      return password;
   }
   @Override
   public String getUsername() {
      return username;
   }
}
```

On remarque que la classe implémente UserDetails dont la documentation est disponible ici :

https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/core/userdetails/UserDetails.html

Cette interface nous fournit des facilités pour la gestion de nos utilisateurs et de leurs rôles dans l'application.

Ajoutez également l'énumération Role dont le code est :

```
public enum Role {
    ADMIN,
    USER,
    MODERATOR
}
```

La seconde étape est de créer une interface Repository qui va faire la liaison avec la base de données et nous « fabriquer » des objets User.

A vous : créez cette interface dans le dossier Repository de votre application

```
import org.springframework.data.jpa.repository.JpaRepository;
import sio.projet.camping.entity.User;

import java.util.Optional;

public interface UserRepository extends JpaRepository<User, Long> {
   Optional<User> findByUsername(String username);
}
```

Ensuite, pour fournir des informations sur l'utilisateur à Sping Security, il va falloir créer la classe CustomUserDetails comme ceci :

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;
import sio.projet.camping.entity.User;
import sio.projet.camping.repository.UserRepository;

@Service
public class CustomUserDetailsService implements UserDetailsService {

   @Autowired
   private UserRepository userRepository;

   @Override
   public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
      User user = userRepository.findByUsername(username)
            .orElseThrow(() -> new UsernameNotFoundException("Utilisateur non trouvé : " +
```

```
username));

      org.springframework.security.core.userdetails.User usr = new
org.springframework.security.core.userdetails.User(
            user.getUsername(),
            user.getPassword(),
            user.getAuthorities() // les rôles et permissions de l'utilisateur
      );

      System.out.println(usr.getAuthorities());

      return usr;
   }
}
```

Lorsque l'utilisateur tente de se connecter, Spring Security appelle loadUserByUsername pour récupérer les informations de cet utilisateur. CustomUserDetailsService est l'endroit où vous allez chercher les données de l'utilisateur (par exemple, dans une base de données) et les transformer en un objet UserDetails que Spring Security peut utiliser pour gérer l'authentification et l'autorisation.

Création de la classe JWTUtils

Cette classe va fournir toutes les fonctionnalités pour générer, parser et valider les jetons JWT. Elle sera rangée dans le package ***security***.

```
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import org.springframework.stereotype.Component;

import java.util.Date;

@Component
public class JwtUtils {

   private final String jwtSecret =
"laclesecretelaclesecretelaclesecretelaclesecretelaclesecretelaclesecretelaclesecretelaclesecrete"; //
Utilisez une clé secrète complexe et unique
   private final int jwtExpirationMs = 86400000; // 1 jour (en millisecondes)

   public String generateJwtToken(String username) {
      return Jwts.builder()
            .setSubject(username)
            .setIssuedAt(new Date())
            .setExpiration(new Date((new Date()).getTime() + jwtExpirationMs))
            .signWith(SignatureAlgorithm.HS512, jwtSecret)
            .compact();
```

```
    }

    public boolean validateJwtToken(String token) {
        try {
            Jwts.parser().setSigningKey(jwtSecret).parseClaimsJws(token);
            return true;
        } catch (Exception e) {
            return false;
        }
    }

    public String getUserNameFromJwtToken(String token) {
        return Jwts.parser()
                .setSigningKey(jwtSecret)
                .parseClaimsJws(token)
                .getBody()
                .getSubject();
    }
}
```

Création du filtre d'analyse du token

Nous créons ici un filtre d'analyse pour récupérer une fois par requête les données sur le token.

Voici le code à placer dans le package ***security*** :

```
import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.web.authentication.WebAuthenticationDetailsSource;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;
import sio.projet.camping.service.CustomUserDetailsService;

import java.io.IOException;

@Component
public class JwtAuthTokenFilter extends OncePerRequestFilter {

    @Autowired
    private JwtUtils jwtUtils;

    @Autowired
    private CustomUserDetailsService userDetailsService;
```

```
   @Override
   protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
FilterChain filterChain)
         throws ServletException, IOException {
      try {
         String jwt = parseJwt(request);
         if (jwt != null && jwtUtils.validateJwtToken(jwt)) {
            String username = jwtUtils.getUserNameFromJwtToken(jwt);

            UserDetails userDetails = userDetailsService.loadUserByUsername(username);
            UsernamePasswordAuthenticationToken authentication =
                  new UsernamePasswordAuthenticationToken(userDetails, null,
userDetails.getAuthorities());
            authentication.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));

            SecurityContextHolder.getContext().setAuthentication(authentication);
         }
      } catch (Exception e) {
         System.out.println("Cannot set user authentication: " + e);
      }

      filterChain.doFilter(request, response);
   }

   private String parseJwt(HttpServletRequest request) {
      String headerAuth = request.getHeader("Authorization");

      if (headerAuth != null && headerAuth.startsWith("Bearer ")) {
         return headerAuth.substring(7);
      }

      return null;
   }
}
```

Mise en place de la configuration de sécurité

Maintenant, nous allons créer la classe SecurityConfig dans le dossier ***security***.

On va y :

- Activer l'authentification stateless.
- Définir les règles d'accès par rôle.
- Appliquer le filtre JWT.

Le code est le suivant :

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationManager;
import
org.springframework.security.config.annotation.authentication.configuration.AuthenticationConfigur
ation;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configurers.AbstractHttpConfigurer;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;
import sio.projet.camping.service.CustomUserDetailsService;

@Configuration
@EnableWebSecurity
public class SecurityConfig {

   private final JwtAuthTokenFilter jwtAuthTokenFilter;
   private final CustomUserDetailsService customUserDetailsService;
   private final PasswordEncoder passwordEncoder;

   public SecurityConfig(JwtAuthTokenFilter jwtAuthTokenFilter,
                         CustomUserDetailsService customUserDetailsService,
                         PasswordEncoder passwordEncoder) {
      this.jwtAuthTokenFilter = jwtAuthTokenFilter;
      this.customUserDetailsService = customUserDetailsService;
      this.passwordEncoder = passwordEncoder;

   }

   @Bean
   public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
      http
            .csrf(AbstractHttpConfigurer::disable) // Désactive CSRF avec la nouvelle syntaxe
            .authorizeHttpRequests(auth -> auth
                  .requestMatchers("/api/auth/**").permitAll() // Permettre l'accès libre aux routes
d'authentification
                  .requestMatchers("/metrics").hasRole("USER") // Autoriser uniquement les USER
                  .requestMatchers("/metrics/all").hasRole("ADMIN") // Autoriser uniquement les USER
                  .anyRequest().authenticated() // Toutes les autres routes doivent être authentifiées
            )
            .sessionManagement(session -> session
                  .sessionCreationPolicy(SessionCreationPolicy.STATELESS) // Gestion de session sans état
            );

      http.addFilterBefore(jwtAuthTokenFilter, UsernamePasswordAuthenticationFilter.class);
```

```
        return http.build();
    }

    @Bean
    public AuthenticationManager authenticationManager(AuthenticationConfiguration authConfig)
throws Exception {
        return authConfig.getAuthenticationManager();
    }
}
```

N'oublions pas le code du contrôleur

```
import java.util.HashMap;
import java.util.Map;



@RestController
@RequestMapping("/api/auth")
public class AuthController {

    @Autowired
    private AuthenticationManager authenticationManager;

    @Autowired
    private JwtUtils jwtUtils;

    @PostMapping("/login")
    public Map<String, String> authenticateUser(@RequestParam String username, @RequestParam
String password) {
        try {
            Authentication authentication = authenticationManager.authenticate(
                    new UsernamePasswordAuthenticationToken(username, password));

            String jwt = jwtUtils.generateJwtToken(username);

            Map<String, String> response = new HashMap<>();
            response.put("token", jwt);
            return response;

        } catch (AuthenticationException e) {
            throw new RuntimeException("Nom d'utilisateur ou mot de passe incorrect");
        }
    }
}
```

Et le code de la classe de configuration pour déclarer le Bean de l'encodeur :

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;

@Configuration
public class AppConfig {
   @Bean
   public PasswordEncoder passwordEncoder() {
      return new BCryptPasswordEncoder();
   }
}
```

A vous :

a. Effectuez quelques tests de l'API avec Postman ou un client REST pour vérifier son bon fonctionnement.

b. Ajoutez des filtres afin de n'autoriser la suppression de log qu'aux administrateurs.

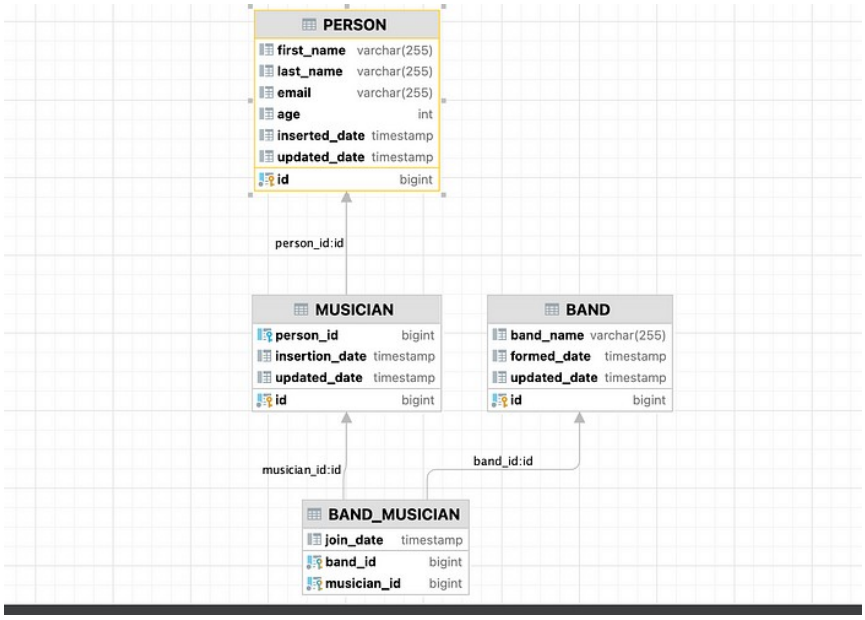
Activité 2 : Utilisation de l'ORM Hibernate

Cette activité consiste à approfondir ce qui a déjà été survolé dans la partie sécurité précédente. Vous y avez manipulé l'ORM Hibernate qui permet de ne plus utiliser de SQL et de manipuler les tables et enregistrements des bases de données comme des objets.

Nous allons reprendre le contexte fourni par un excellent tutorial disponible sur le site suivant :

https://medium.com/@phoenixrising_93140/springboot-with-hibernate-a-restful-example-d367ad50b7f1

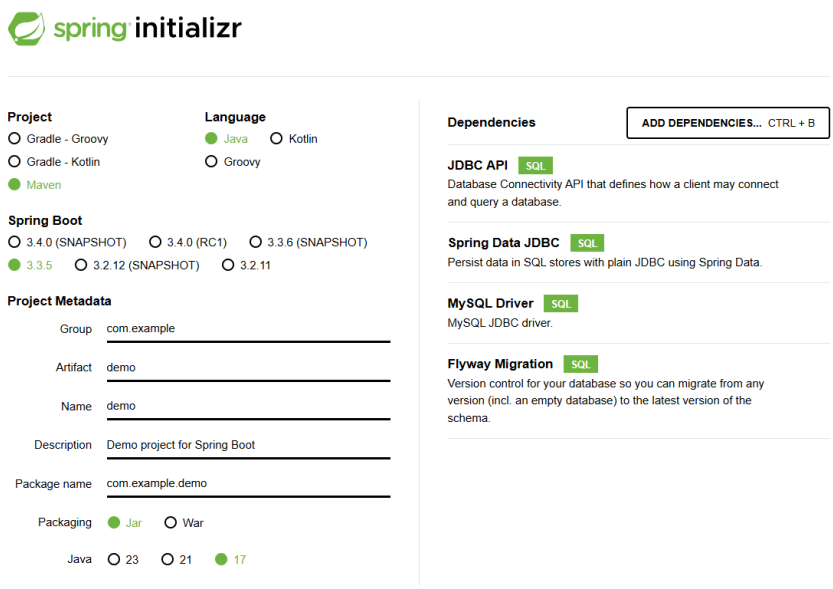
Il consiste à découvrir l'ORM Hibernate via la réalisation d'une API spring boot de gestion de données autour de groupes de musique. Le modèle relationnel que nous allons implémenter est le suivant :



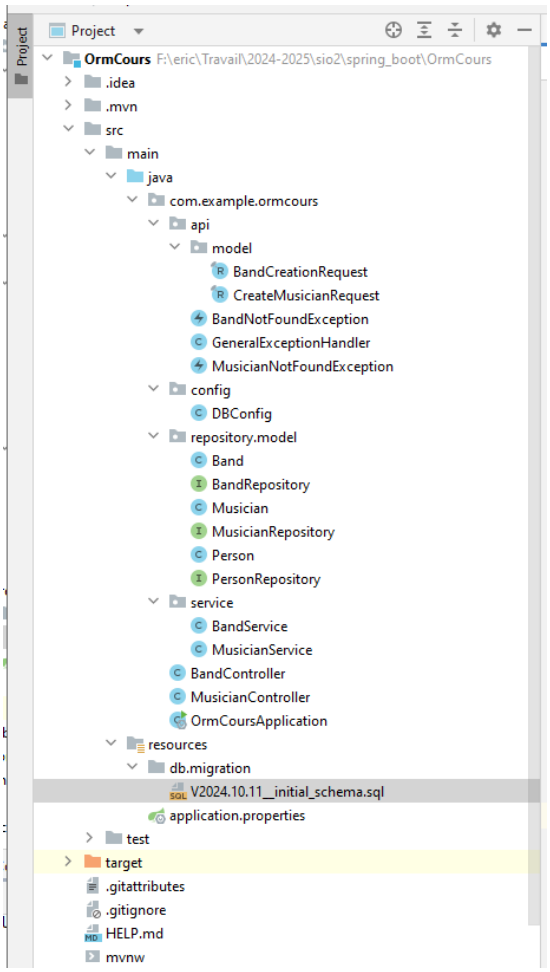
Nous avons des personnes qui peuvent être des musiciens. Chaque musicien est une personne et joue dans un ou plusieurs groupes (Band). La table BAND_MUSICIAN est l'association entre les tables musicien et groupe. Nous avons donc ici des associations de un à plusieurs et de plusieurs à plusieurs (0.1 – 0.n et 0.n – 0.n sur les MCD). Nous allons structurer notre code pour gérer cela.

Création du projet :

Créez un nouveau projet spring boot avec les dépendances suivantes :



Le projet sera structuré de cette manière, nous allons voir chacun des éléments.



Le code vous est donné dans une archive jointe, vous l'utiliserez pour créer votre projet étape par étape de votre côté.

Création de la base de données

La première chose à remarquer se trouve dans le dossier resource. Il s'agit du dossier db.migration qui va contenir le (les) scripts de création de notre base.

Observez le contenu, vous y verrez des ordres SQL de création de nos tables.

```
CREATE TABLE PERSON
(
   id          BIGINT PRIMARY KEY AUTO_INCREMENT,
   first_name    VARCHAR(255),
   last_name     VARCHAR(255),
   email        VARCHAR(255),
   age          INT,
   inserted_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
   updated_date  TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);

-- if we include band id here then it means a musician belongs to only one band and that is not the case.
CREATE TABLE MUSICIAN
(
   id          BIGINT PRIMARY KEY AUTO_INCREMENT,
   person_id     BIGINT,
   insertion_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
   updated_date   TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
   FOREIGN KEY (person_id) REFERENCES PERSON (id)
);

CREATE TABLE BAND
(
   id         BIGINT PRIMARY KEY AUTO_INCREMENT,
   band_name    VARCHAR(255),
   formed_date  TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
   updated_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);

-- This is required to model many to many relationship between band and musician
CREATE TABLE BAND_MUSICIAN
(
   band_id     BIGINT,
   musician_id BIGINT,
   join_date   TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
   PRIMARY KEY (band_id, musician_id),
   FOREIGN KEY (band_id) REFERENCES BAND (id),
   FOREIGN KEY (musician_id) REFERENCES MUSICIAN (id)
);
```

Ce script sera exécuté par une dépendance nommée Flyway qui va automatiquement gérer les versions de nos scripts SQL. Nous n'entrerons pas dans les détails, mais disons qu'il va faciliter la gestion de la structure de notre base afin de la maintenir à jour plus facilement.

Vous trouverez dans le fichier de configuration application.properties les informations pour votre base de données et pour flyway justement. Voici le contenu :

```
spring.application.name=OrmCours

# Database Configuration
spring.datasource.url=jdbc:mysql://localhost:3306/bandgroup
spring.datasource.username=root
spring.datasource.password=
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
spring.jpa.hibernate.ddl-auto=validate

# Flyway Configuration
spring.flyway.url=jdbc:mysql://localhost:3306/bandgroup?characterEncoding=UTF-
8&useLegacyDatetimeCode=false&serverTimezone=UTC&useSSL=false&allowPublicKeyRetrieval=true
spring.flyway.user=root
spring.flyway.password=
spring.flyway.locations=classpath:db.migration
```

Le concept d'entity

Rappelez vous vos cours de modélisation de données, on vous fait construire des MCD dans lesquels vous avez des entités et des associations. Ces entités sont ce qui deviendra ensuite vos tables (avec les associations n vers n). Ici, les entités sont des classes Java que l'on va annoter afin que l'ORM Hibernate puisse faire le lien avec la table qu'il représente. Nous allons donc créer les entités nécessaires à notre programme.

L'entity Person

On commence par l'annoter avec Table et lui spécifier le nom de la table concernée.

```
@Entity
@Table(name = "PERSON")
public class Person {
```

Ensuite, on s'intéresse à ses champs eux aussi annotés de manière à coller à la structure de la base de données.

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;

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

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

public List<Musician> getMusicians() {
   return musicians;
}

public void setMusicians(List<Musician> musicians) {
   this.musicians = musicians;
}

@Column(name = "email")
private String email;
```

@Column permet de mapper l'attribut avec le champ dont on donne le nom dans la table de notre BDD Mysql.

On remarque une getter/setter sur une collection de musicians. Elle est annotée en dessous de la manière suivante :

```
@OneToMany(mappedBy = "person")
@JsonIgnore //needed to avoid infinite recursion
private List<Musician> musicians;
```

@OneToMany indique que notre table mappée sera à l'origine d'une clé étrangère spécifiée dans la table Musician. Cette relation permettra de récupérer à partir d'une personne tous les musiciens qui pointent vers elle. On considère ici qu'une personne pourra donner lieu à plusieurs musiciens différents dans la base (par exemple, on pourrait créer 2 musiciens si une personne joue à la fois de la guitare et du piano, un par instrument). De même, on a créé Person car il est possible qu'on mémorise des personnes dans la base qui ne sont pas des musiciens (relation 0,n de personne vers musicien).

Avec cette annotation, le système ira rechercher toutes les instances de ***musician*** en relation avec notre ***person***.

L'entity Musician

Du coté de cette entité, nous allons représenter le musicien. Même principe, nous annotons pour faire le lien avec sa table en base de données et spécifions les attributs en cohérence avec celle-ci.

```
@Entity
@Table(name = "MUSICIAN")
public class Musician {

   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;

   @ManyToOne(cascade = CascadeType.PERSIST)
   @JoinColumn(name = "person_id")
   private Person person;

   @ManyToMany(mappedBy = "musicians")
   @JsonIgnore //needed to avoid infinite recursion
   private List<Band> bands = new ArrayList<>();

   public Musician() {
   }
```

La différence ici vient de la gestion de la clé étrangère vers Person. On a en effet une relation de un à un (cardinalité 1,1). On spécifie @ManyToOne pour l'attribut person qui est une instance de la classe Person. L'attribut de l'annotation « cascade » permet de spécifier le comportement à avoir vis-à-vis de l'enregistrement associé dans Person. (lire : https://www.geeksforgeeks.org/hibernate-different-cascade-types/)

Il faut remarque la gestion de l'association de plusieurs à plusieurs (@ManyToMany). Du coté du musicien, on indique qu'on a un lien avec la classe du groupe dans lesquels il joue (List de Band). L'attribut de l'annotation mappedBy nous donne une indication sur le chemin inverse lorsqu'on est dans Band. Il faudra passer par la collection « musicians » depuis Band pour trouver les musiciens qui le composent.

L'entité Band

Du coté de Band :

```
@Entity
@Table(name = "BAND")
public class Band {

   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;

   @Column(name = "band_name")
   private String bandName;

   public Band(String bandName) {
      this.bandName = bandName;
   }
```

Pas de difficultés ici. Passons à la gestion de la relation vers Musician :

```
@ManyToMany
@JoinTable(
      name = "BAND_MUSICIAN",
      joinColumns = @JoinColumn(name = "band_id"),
      inverseJoinColumns = @JoinColumn(name = "musician_id")

)
private List<Musician> musicians = new ArrayList<>();
```

On spécifie ici la @JoinTable en lui indiquant son nom et on définit les colonnes liées, dans un sens, et dans l'autre. Grace à tout cela, le système d'ORM va être capable de gérer cette relation.

A noter que si l'association BAND_MUSICIAN avait nécessité des champs supplémentaires dans la base de données, il aurait fallu créer une Entity à part entière et décrire la table en spécifiant les attributs comme nous l'avons fait pour les entités précédentes.

Les repository

Les entités sont créées. Il nous faut maintenant pour chacun d'entre elles un repository. Il s'agit de la classe qui va faire le lien entre nos entités et la base de données. C'est depuis le repository qu'on pourra appeler les méthodes CRUD ainsi que des méthodes personnalisées sur nos données.

Avec les interfaces fournies par Spring Data, telles que JpaRepository, CrudRepository, ou PagingAndSortingRepository, un repository permet d'exécuter des opérations CRUD de manière simple et rapide :

- **Save** une nouvelle entité ou mettre à jour une entité existante : save()
- **Trouver** une entité par son ID : findById()
- **Récupérer** toutes les entités : findAll()
- **Supprimer** une entité : deleteById()

Spring Data implémente automatiquement ces méthodes pour toutes les entités, ce qui réduit considérablement le besoin d'écrire du code SQL.

Voyons comment écrire cela pour chacun de nos besoins.

Le repository de Person

Ajoutez la classe PersonRepository suivante dans le dossier Entity de votre projet :

```
import org.springframework.data.jpa.repository.JpaRepository;

public interface PersonRepository extends JpaRepository<Person, Long> {
}
```

Si nous ne voulons que des méthodes CRUD par défaut, le JpaRepository est déjà capable de nous les fournir sans rien coder. On spécifie juste le Type d'Entity concerné et le type de son identifiant (ici <Person, Long>).

Le repository de Musician

Pour l'accès aux données des musiciens, nous aurons ce repository :

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;

import java.util.List;

public interface MusicianRepository extends JpaRepository<Musician, Long> {

   @Query("select b from Band b join b.musicians m where m.id = ?1")
   List<Band> findAllBandsByMusicianId(long musicianId);

}
```

Ici, on a ajouté une méthode annotée @Query qui permet de créer un accès personnalisé aux données (ici tous les groupes ou joue un musicien donné).

Le repository de Band

Pour les groupes, voici à présent le code du repository :

```
import org.springframework.data.jpa.repository.JpaRepository;

public interface BandRepository extends JpaRepository<Band, Long> {
}
```

Les services

Nous avons maintenant nos Entity et les Repository qui permettent l'interaction avec les données réelles de la base. Pour orchestrer tout cela, nous pouvons soit coder des appels aux Repository dans nos contrôleurs, soit passer par des classes dédiées à cela qu'on appellera des Services. Ainsi, le contrôleur fera appel à ces services auxquels il déléguera la logique d'interaction avec la base de données.

Nous créerons 2 services pour le moment :

- Un pour les musiciens : MusicianService
- Un pour les groupes de musique : BandService

La classe MusicianService

Cette classe aura le code suivant :

```
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.Optional;

@Service
public class MusicianService {


   private final MusicianRepository musicianRepository;

   private final PersonRepository personRepository;

   public MusicianService(MusicianRepository musicianRepository, PersonRepository personRepository) {
      this.musicianRepository = musicianRepository;
      this.personRepository = personRepository;
   }

   public Musician createMusician(CreateMusicianRequest createMusicianRequest) {
      return musicianRepository.save(mapToMusician(createMusicianRequest));
   }

   public Optional<Musician> getMusician(long id) {
      return musicianRepository.findById(id);
   }

   private Musician mapToMusician(CreateMusicianRequest createRequest) {
      Musician musician = new Musician();
      Person person = new Person();
      person.setFirstName(createRequest.firstName());
      person.setLastName(createRequest.lastName());
      person.setEmail(createRequest.email());
      person.setAge(createRequest.age());
      personRepository.save(person);
```

```
        musician.setPerson(person);
        return musician;
    }

    public List<Band> getBandsByMusicianId(long musicianId) {
        return musicianRepository.findAllBandsByMusicianId(musicianId);
    }

}
```

La classe BandService

Cette classe sera codée comme ceci :

```
import com.example.ormcours.api.model.BandCreationRequest;
import com.example.ormcours.repository.model.Band;
import com.example.ormcours.repository.model.BandRepository;
import com.example.ormcours.repository.model.MusicianRepository;
import jakarta.transaction.Transactional;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.Optional;

@Service
public class BandService {

    private final BandRepository bandRepository;

    private final MusicianRepository musicianRepository;

    public BandService(BandRepository bandRepository, MusicianRepository musicianRepository) {
        this.bandRepository = bandRepository;
        this.musicianRepository = musicianRepository;
    }

    public boolean isHealthy() {
        return bandRepository.count() >= 0;
    }

    public Band createBand(BandCreationRequest bandCreationRequest) {
        return bandRepository.save(new Band(bandCreationRequest.bandName()));
    }

    public Optional<Band> getBand(final long id) {
        return bandRepository.findById(id);
    }

    @Transactional
    public Optional<Band> addMusiciansToBand(long bandId, List<Long> musicianIds) {
        return bandRepository.findById(bandId)
                .map(band -> {
                    band.setMusicians(musicianRepository.findAllById(musicianIds));
                    return bandRepository.save(band);
                });
    }

}
```

Le @transactional permet de mettre le contenu de la méthode dans une transaction SQL. Ainsi, on s'assurera que si un problème survient, l'ensemble des opérations effectuées sera remis à l'état initial tel qu'il était avant le début de la méthode.

Les contrôleurs

Il nous reste à créer les contrôleurs pour fournir nos routes d'API. Nous en ferons 2 :

MusicianController et BandController. Vous êtes déjà familiarisé avec ce type de fichier. Reproduisez le code pour faire fonctionner votre API.

MusicianController

Le code est le suivant :

```
@RestController
@RequestMapping("/api/v1/musicians")
public class MusicianController {

   private final MusicianService musicianService;

   public MusicianController(MusicianService musicianService) {
      this.musicianService = musicianService;
   }

   @PostMapping("/test")
   public String test(){
      return "coucou";
   }

   @PostMapping
   public Musician createMusician(@RequestBody CreateMusicianRequest createMusicianRequest) {
      return musicianService.createMusician(createMusicianRequest);
   }

   @GetMapping("/{id}")
   public Musician getMusician(@PathVariable long id) throws MusicianNotFoundException {
      return musicianService.getMusician(id).orElseThrow(() -> new MusicianNotFoundException());
   }

   @GetMapping("/bands/{musicianId}")
   public List<Band> getBands(@PathVariable long musicianId) {
      return musicianService.getBandsByMusicianId(musicianId);
   }

}
```

BandController

Le code est le suivant :

```
@RestController
@RequestMapping("/api/v1/bands")
public class BandController {

   private final BandService bandService;

   public BandController(BandService bandService) {
      this.bandService = bandService;
   }
/*
   @GetMapping("/health")
   public HealthStatus getHealthStatus() {
      return new HealthStatus(bandService.isHealthy());
   }

 */

   @PostMapping
   public Band createBand(@RequestBody BandCreationRequest bandCreationRequest) {
      return bandService.createBand(bandCreationRequest);
```

```
    }

    @GetMapping("/{id}")
    public Band getBand(@PathVariable long id) throws BandNotFoundException {
        return bandService.getBand(id).orElseThrow(() -> new BandNotFoundException());
    }

    @PatchMapping("/{bandId}/musicians")
    public Band addMusiciansToBand(@PathVariable long bandId, @RequestBody List<Long> musicianIds) throws BandNotFoundException {
        return bandService.addMusiciansToBand(bandId, musicianIds).orElseThrow(() -> new BandNotFoundException());
    }

}
```

A vous :

Vous allez faire évoluer l'API en ajoutant :

a. Pour chacun des musiciens, on devra connaître l'instrument dont il joue (Guitare, Piano, Batterie, Chant …). S'il en joue plusieurs, on devra créer 2 enregistrements dans base mais ils seront rattachés à une seule et même personne.

b. On souhaite connaître la date d'entrée dans un groupe pour un musicien. De même, lorsqu'il quitte le groupe, il faudra ajouter la date à laquelle il en est parti. Modifiez les structures pour prendre cela en compte et testez-le.

On souhaite ajouter les festivals auxquels les groupes participent. On a donc besoin d'ajouter une table festival (id, libelle, ville, date, tarif) et de mémoriser les groupes qui y participent.

c. Modifiez le code pour prendre cela en compte et testez l'API.