Spring Data JPA — Accès aux bases de données

11. Introduction à Spring Data JPA et à l'ORM

La plupart des applications nécessitent de sauvegarder des données de manière persistante. Bien que vous puissiez écrire des requêtes SQL manuelles (via JDBC), Spring propose une approche beaucoup plus moderne et puissante : **Spring Data JPA**.

JPA (Java Persistence API) utilise le concept d'**ORM (Object-Relational Mapping)**. L'idée est simple : au lieu de manipuler des tables et des colonnes SQL, vous manipulez directement vos classes et objets Java. Le framework (généralement *Hibernate*, qui est l'implémentation par défaut de JPA dans Spring) se charge de traduire automatiquement ces objets en requêtes SQL en arrière-plan.

12. Configuration de la connexion à MySQL

Pour commencer, vous devez ajouter deux dépendances dans votre fichier `pom.xml` (si vous utilisez Maven) ou via Spring Initializr :

- **Spring Data JPA**: Le module principal.
- **MySQL Driver**: Le pilote permettant à Java de communiquer avec une base MySQL.

Ensuite, ouvrez le fichier de configuration `src/main/resources/application.properties` pour y définir les paramètres de votre base de données :

```
# Paramètres de connexion MySQL
spring.datasource.url=jdbc:mysql://localhost:3306/mystery_db?serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=votre_mot_de_passe

# Afficher les requêtes SQL générées dans la console (très utile pour le débogage)
spring.jpa.show-sql=true

# Comportement de la création des tables (Génération automatique)
# "update" crée les tables si elles n'existent pas et les met à jour si vous modifiez vos
classes
spring.jpa.hibernate.ddl-auto=update

# Dialecte SQL spécifique à MySQL
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
```

13. Création des Entités (Entities) : Mapper Java vers SQL

Une **Entité** est une simple classe Java décorée d'annotations spécifiques qui indiquent à Spring comment créer la table correspondante dans MySQL.

Créons une entité pour gérer les données de notre investigation. Pour assurer une architecture robuste dès le premier jour, nous intégrons directement le support de localisation (Anglais et Arabe) ainsi que les métadonnées visuelles nécessaires pour un affichage sur un tableau de preuves (corkboard).

```
package com.studiogames.mysterybackend.model;

import jakarta.persistence.*;

// @Entity indique que cette classe sera une table dans la base de données
@Entity
// @Table permet de personnaliser le nom de la table (optionnel)
@Table(name = "suspects")
public class Suspect {

    // @Id définit la clé primaire (Primary Key)
    @Id
    // @GeneratedValue indique que MySQL va auto-incrémenter l'ID (1, 2, 3...)
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    // Prise en charge de la localisation (Arabe + Anglais) dès la conception
    @Column(name = "name_en", nullable = false)
    private String nameEn;

    @Column(name = "name_ar", nullable = false, columnDefinition = "VARCHAR(255) CHARACTER
SET utf8mb4 COLLATE utf8mb4_unicode_ci")
    private String nameAr;

    @Column(length = 1000)
    private String alibi;

    // Métadonnées pour l'interface utilisateur (Esthétique Dark Noir)
    @Column(name = "pin_color")
    private String pinColor = "AMBER"; // Couleur de la punaise sur le tableau d'indices

    @Column(name = "is_pinned_to_corkboard")
    private boolean pinnedToCorkboard = false;

    // Constructeurs
    public Suspect() {}

    public Suspect(String nameEn, String nameAr, String alibi) {
        this.nameEn = nameEn;
        this.nameAr = nameAr;
        this.alibi = alibi;
    }

    // Getters et Setters (Omis ici pour la concision, mais obligatoires)
    // ...
}
```

La magie de l'ORM : Au lancement de l'application (grâce à `ddl-auto=update`), Spring Boot va lire cette classe et exécuter automatiquement une requête `CREATE TABLE suspects (...)` dans

votre MySQL, en configurant correctement les types de colonnes (VARCHAR, BIGINT, BOOLEAN) et l'encodage (utf8mb4 pour l'Arabe).

14. Le Repository : L'interface magique pour le CRUD

C'est ici que Spring Data JPA brille véritablement. Pour effectuer des opérations **CRUD** (Create, Read, Update, Delete) sur notre table, nous n'avons pas besoin d'écrire de code complexe. Il suffit de créer une interface (Interface Java) qui hérite de `JpaRepository`.

```
package com.studiogames.mysterybackend.repository;

import com.studiogames.mysterybackend.model.Suspect;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import java.util.List;

@Repository
public interface SuspectRepository extends JpaRepository<Suspect, Long> {

    // Vous bénéficiez instantanément des méthodes : save(), findAll(), findById(),
deleteById().

    // Spring Data permet aussi de créer des requêtes personnalisées juste en nommant bien
la méthode !
    // Ex: Trouver tous les suspects épinglés sur le tableau
    List<Suspect> findByPinnedToCorkboardTrue();
}
```

15. Exécuter les opérations CRUD depuis le Contrôleur

Nous allons maintenant injecter (Dependency Injection) notre `SuspectRepository` dans notre contrôleur REST pour connecter notre base de données aux requêtes HTTP du client.

```
package com.studiogames.mysterybackend.controller;

import com.studiogames.mysterybackend.model.Suspect;
import com.studiogames.mysterybackend.repository.SuspectRepository;
import org.springframework.web.bind.annotation.*;
import org.springframework.http.ResponseEntity;

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

@RestController
@RequestMapping("/api/suspects")
public class SuspectDatabaseController {
```

```
    private final SuspectRepository suspectRepository;

    // Injection de dépendance via le constructeur
    public SuspectDatabaseController(SuspectRepository suspectRepository) {
        this.suspectRepository = suspectRepository;
    }

    // CREATE (Insérer en base de données)
    @PostMapping
    public Suspect createSuspect(@RequestBody Suspect suspect) {
        // La méthode save() génère automatiquement le SQL 'INSERT INTO'
        return suspectRepository.save(suspect);
    }

    // READ (Récupérer tous les éléments)
    @GetMapping
    public List<Suspect> getAllSuspects() {
        // La méthode findAll() génère le SQL 'SELECT * FROM'
        return suspectRepository.findAll();
    }

    // UPDATE (Mettre à jour un élément existant)
    @PutMapping("/{id}")
    public ResponseEntity<Suspect> updateSuspect(@PathVariable Long id, @RequestBody Suspect
updatedData) {
        Optional<Suspect> existingSuspect = suspectRepository.findById(id);

        if (existingSuspect.isPresent()) {
            Suspect suspect = existingSuspect.get();
            suspect.setNameEn(updatedData.getNameEn());
            suspect.setNameAr(updatedData.getNameAr());
            suspect.setPinnedToCorkboard(updatedData.isPinnedToCorkboard());

            // save() agit comme un 'UPDATE' si l'objet possède déjà un ID
            return ResponseEntity.ok(suspectRepository.save(suspect));
        } else {
            return ResponseEntity.notFound().build();
        }
    }

    // DELETE (Supprimer un élément)
    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteSuspect(@PathVariable Long id) {
        if (suspectRepository.existsById(id)) {
            suspectRepository.deleteById(id); // Génère le SQL 'DELETE FROM'
            return ResponseEntity.noContent().build();
        }
        return ResponseEntity.notFound().build();
    }
}
```

*Rédigé et structuré par : **Med Khalil Kribi***

Source : https://apcpedagogie.com/spring-data-jpa-acces-base-de-donnees · téléchargé le 07/11/2026 · © APC Pédagogie · apcpedagogie.com