**INTRODUCTION GÉNÉRALE**

Ce projet est une **application de gestion des employés (Employee Management System ).**  
Elle permet par exemple :

* **D’afficher** la liste des employés,
* **d’ajouter** un nouvel employé,
* De **modifier** ou **supprimer** un employé.

- Le **backend (Spring Boot)** :

Gère **les données, la logique métier et la communication avec la base de données**.  
- Le **frontend (React.js)**

Gère **l’interface utilisateur** et communique avec le backend via des **API REST**.

**PARTIE 1: LE BACKEND (ems-backend)**

Le backend suit l’architecture **MVC (Model-View-Controller)** et contient plusieurs couches bien séparées :

**Structure globale**

|  |
| --- |
| ems-backend/  ├── controller/ EmployeeController.java  ├── dto/ EmployeeDto.java  ├── entity/ Employee.java  ├── exception/ RessourceNotFoundException.java  ├── mapper/ EmployeeMapper.java  ├── repository/ EmployeeRepository.java  ├── service/  │ ├── EmployeeService.java  │ └── EmployeeServiceImpl.java  └── resources/ application.properties |

**1. entity/ (Les entités)**

* **But :** C’est la **représentation de la table** dans la base de données sous forme de classes Java.

**Employee.java** contient des champs comme id, firstName, lastName, email.

Chaque attribut correspond à une colonne dans la table employees.  
On y met aussi les annotations **JPA** (@Entity, @Table, @Id, @GeneratedValue, etc.) pour que Spring comprenne comment stocker ces objets dans la base de données.

| **Annotation** | **Description** |
| --- | --- |
| **@Entity** | Indique à JPA que cette classe correspond à une **table**. |
| **@Table**(name = "employees") | Donne le nom de la table SQL. |
| **@Id** | Indique la **clé primaire**. |
| **@GeneratedValue**(strategy = GenerationType.IDENTITY) | L’ID est généré automatiquement (auto-incrément). |
| **@Column**(name = "first\_name") | Personnalise le nom de la colonne. |
| **nullable = false** | Rend la colonne obligatoire (non nulle). |
| **Annotation** | Signification |
| **@Getter** | Génère automatiquement toutes les méthodes get (ex : getId()...) |
| **@Setter** | Génère automatiquement toutes les méthodes set (ex : setId()) |
| **@NoArgsConstructor** | Génère un constructeur vide. |
| **@AllArgsConstructor** | Génère un constructeur avec tous les attributs. |

**2. dto/ Data Transfer Object**

* **But :** transmettre les données entre le backend et le frontend **sans exposer directement l’entité de la base**.

Autrement dit c’est une version simplifiée de l’employé utilisée pour échanger les données avec le frontend.

**EmployeeDto.java** est une version simplifiée de Employee qu’on envoie ou reçoit via les requêtes HTTP.

* Cela évite de renvoyer des champs sensibles ou inutiles à l’utilisateur.

**3. mapper/ (Conversion Entity ↔ DTO)**

* **But :** convertir facilement un Employee (entité) en EmployeeDto, et inversement.

**EmployeeMapper.java** contient des méthodes comme :

|  |
| --- |
| public static **EmployeeDto** **mapToEmployeeDto**(Employee employee)  public static **Employee** **mapToEmployee**(EmployeeDto employeeDto) |

* C’est une **bonne pratique** pour séparer la couche métier de la couche de présentation.

Autrement dit on sépare les deux pour **ne jamais exposer directement la structure de la base de données**.

**4. repository/ Accès à la base de données**

* **But :** communiquer avec la base via **Spring Data JPA**.

**EmployeeRepository.java** hérite de JpaRepository<Employee, Long>.

Grâce à cela, on peut directement utiliser des méthodes comme : **findAll(), findById(), save(), deleteById()** sans écrire de SQL manuellement.

**5. service/ Logique métier**

* **But :** centraliser la logique métier, entre le contrôleur et la base de données. (gère la **logique métier**.)

**EmployeeService.java :** c’est **une interface** qui définit les opérations disponibles (ajouter, modifier, supprimer, lister...).

**EmployeeServiceImpl.java** c’est **l’implémentation** de cette interface, où on écrit la logique réelle :

* Vérification si un employé existe,
* Gestion des exceptions,
* Appel au repository.

Cela permet de séparer la **définition du contrat** (interface) et **l’implémentation réelle**.

| **Élément** | **Explication** |
| --- | --- |
| **@Service** | Indique à Spring que c’est un **service** (composant métier). |
| **@AllArgsConstructor** | Génère un constructeur pour injecter EmployeeRepository. |

**Exemple:**

* **Methode createEmployee:**

|  |
| --- |
| public EmployeeDto **createEmployee**(EmployeeDto employeeDto) {  Employee employee = EmployeeMapper.mapToEmployee(employeeDto);  Employee savedEmployee = **employeeRepository**.save(employee);  return EmployeeMapper.mapToEmployeeDto(savedEmployee);  } |

**Étapes:**

1. On convertit le **DTO reçu du frontend** en **entité** (mapToEmployee).
2. On **enregistre** dans la **base** (save).
3. On **reconvertit** en **DTO** pour la **réponse** (mapToEmployeeDto).

**La méthode getAllEmployees()**

|  |
| --- |
| List<Employee> employees = employeeRepository.findAll();  return employees.**stream()**  .**map**(EmployeeMapper::mapToEmployeeDto)  .**collect**(Collectors.toList()); |

* **stream() → transforme la liste en flux.**
* **map() → applique une fonction sur chaque élément (ici, conversion en DTO).**
* **collect() → reforme la liste finale.**

**6. controller/ API REST**

* **But :** exposer des **endpoints REST** accessibles depuis le frontend. Autrement dit, ce fichier gère toutes les **requêtes HTTP** venant du frontend (React). Il expose les **endpoints REST API** : /api/employees.

**EmployeeController.java** contient des méthodes comme :

**@GetMapping**("/employees") → récupérer la liste des employés

**@PostMapping**("/employees") → ajouter un nouvel employé

**@PutMapping**("/employees/{id}") → modifier un employé

**@DeleteMapping**("/employees/{id}") → supprimer un employé

Chaque **méthode** communique avec le **EmployeeService** et renvoie une réponse **JSON** au **frontend**.

|  |
| --- |
| private **EmployeeService** employeeService; |

* **Injection automatique** du service (Spring s’en occupe grâce à **@AllArgsConstructor**).

**Les méthodes du contrôleur:**

* **Créer un employé:**

|  |
| --- |
| **@PostMapping**  public ResponseEntity<EmployeeDto> **createEmployee** (@RequestBody EmployeeDto employeeDto) |

| **Élément** | **Rôle** |
| --- | --- |
| **@PostMapping** | Indique que cette méthode répond à une requête **HTTP POST**. |
| **@RequestBody** | Dit à Spring de **convertir le JSON reçu** depuis le frontend en objet EmployeeDto. |
| **ResponseEntity** | Permet de personnaliser la **réponse HTTP** (corps + code de statut). |
| **HttpStatus.CREATED** | Retourne le code 201, qui signifie "Créé avec succès". |

| **Annotation** | **Signification** |
| --- | --- |
| **@CrossOrigin** ("\*") | **Autorise les requêtes** venant d’autres origines (ex: React sur localhost:5173). Le **\*** veut dire “tous les domaines sont autorisés”. Sans ça, le frontend n’aurait pas le droit d’appeler l’API (erreur CORS). |
| **@AllArgsConstructor** | Génère automatiquement un **constructeur** avec tous les attributs (grâce à **Lombok**). |
| **@RestController** | Indique que cette classe est un **contrôleur REST** → elle renvoie des données **JSON** et **non** des pages **HTML**. |
| **@RequestMapping**("/api/employees") | Définit **l’URL** de base pour toutes les routes de cette classe.  **Exemple:** **/api/employees**, **/api/employees/{id}**. |

* **Récupérer un employé par ID**

|  |
| --- |
| @GetMapping("/{id}")  public ResponseEntity<EmployeeDto> **getEmployeeById**(@PathVariable("id") Long employeeId) |

* **Récupérer tous les employés**

|  |
| --- |
| @GetMapping  public **ResponseEntity**<List<EmployeeDto>> getAllEmployees() |

* Renvoie une **liste JSON de tous les employés**.
* **Mettre à jour un employé**

|  |
| --- |
| @PutMapping("/{id}")  public ResponseEntity<EmployeeDto> **updateEmployee**(@PathVariable("id") Long employeeId, @RequestBody EmployeeDto employeeDto) |

**@PutMapping → méthode** HTTP PUT (modification).

**@PathVariable** → **récupère** l’ID à modifier.

**@RequestBody** → **récupère** le nouvel objet JSON envoyé.

* **Supprimer un employé**

|  |
| --- |
| @DeleteMapping("/{id}")  public **ResponseEntity**<String> **deleteEmployee**(@PathVariable("id") Long employeeId) |

* Renvoie une réponse "Employee deleted successfully" avec un code **200 OK**.

**7. exception/ Gestion des erreurs**

* **But :** gérer les erreurs (exemple : un employé inexistant).

**RessourceNotFoundException.java** est levée quand on ne trouve pas une ressource dans la base :

**throw new RessourceNotFoundException**("Employee not found with id: " + id);

(Si cette exception est levée, la réponse HTTP sera 404 (Not Found).)

→ Elle hérite de **RuntimeException**, donc on peut la lancer sans la déclarer dans la signature (throws).

* Cela évite d’envoyer des erreurs techniques au frontend.

**8. resources/application.properties**

* **But :** fichier de configuration du projet.  
  On y met :

|  |
| --- |
| spring.datasource.url=jdbc:mysql://localhost:3306/ems\_db  spring.datasource.username=root  spring.datasource.password=...  spring.jpa.hibernate.ddl-auto=update  server.port=8080 |

Cela configure la base MySQL, le port du serveur, etc.

| **Propriété** | **Signification** |
| --- | --- |
| spring.datasource.**url** | **Adresse** de la base MySQL. |
| spring.datasource.**username** | **Nom** d’utilisateur MySQL. |
| spring.datasource.**password** | **Mot** de passe MySQL. |
| **hibernate.dialect** | **Type de SQL** adapté à MySQL 8. |
| **ddl-auto=update** | **JPA crée/modifie automatiquement** les tables selon les **entités**. |

**Resume de backend:**

|  |
| --- |
| **React Frontend**  **↓ (HTTP JSON)**  **EmployeeController**  **↓**  **EmployeeServiceImpl**  **↓**  **EmployeeRepository**  **↓**  **Base MySQL** |

**PARTIE 2 : LE FRONTEND — ems-frontend**

Le frontend est développé avec **React.js (JavaScript)** et permet à l’utilisateur de **voir, ajouter, modifier ou supprimer** des employés.

**Structure globale**

|  |
| --- |
| ems-frontend/  ├── src/  │ ├── components/  │ │ ├── EmployeeComponent.jsx  │ │ ├── FooterComponent.jsx  │ │ ├── HeaderComponent.jsx  │ │ └── ListEmployeeComponent.jsx  │ ├── service/ EmployeeService.js  │ ├── App.jsx  │ ├── App.css  │ ├── index.css  │ └── main.jsx  └── public/ index.html |

**1. components/**

Ce dossier contient les **composants React** représentant chaque partie de l’interface.

**ListEmployeeComponent.jsx**

* Affiche la **liste des employés** sous forme de tableau.
* Fait appel au backend via EmployeeService pour récupérer les données.

**EmployeeComponent.jsx**

* Formulaire pour **ajouter ou modifier** un employé.
* Utilise useState pour gérer les champs du formulaire.
* Envoie les données au backend avec EmployeeService.

**HeaderComponent.jsx**

* Barre de navigation ou titre principal de l’application.

**FooterComponent.jsx**

* Pied de page affiché en bas de l’application.

**2. service/EmployeeService.js**

* Contient toutes les **requêtes HTTP** vers le backend (API REST).
* **axios** pour **communiquer** avec le **backend**.

|  |
| --- |
| import axios from "axios";  const **REST\_API\_BASE\_URL** = 'http://localhost:8080/api/employees';  // Liste de tous les employés  export const **listEmployees** = () => axios.get(**REST\_API\_BASE\_URL**);  // Créer un nouvel employé  export const **createEmployee** = (employee) => axios.post(**REST\_API\_BASE\_URL**, employee);  // Récupérer un employé par ID  export const **getEmployee** = (employeeId) => axios.get(**REST\_API\_BASE\_URL** + '/' + employeeId);  // Mettre à jour un employé  export const **updateEmployee** = (employeeId, employee) => axios.put(**REST\_API\_BASE\_URL** + '/' + employeeId, employee);  // Supprimer un employé  export const **deleteEmployee** = (employeeId) => axios.delete(**REST\_API\_BASE\_URL** + '/' + employeeId); |

* axios.get() → pour **lire**
* axios.post() → pour **ajouter**
* axios.put() → pour **modifier**
* axios.delete() → pour **supprimer**

Chaque fonction renvoie une **promesse (Promise)**, donc dans React on utilise .then() et .catch().

Ainsi, le code React reste propre et lisible.

**3. App.jsx, App.css, index.css**

* **App.jsx** : Composant principal où on définit les **routes** (par exemple avec react-router-dom) pour naviguer entre les pages.
* **App.css** et **index.css** : Styles globaux de l’application (mise en forme, couleurs, marges…).

**4. main.jsx et index.html**

* **main.jsx** : point d’entrée de React, où l’application est montée dans le DOM.
* **ReactDOM.createRoot**(document.getElementById('root')).render(<App />);
* **index.html** : structure HTML de base (le div id="root" où React injecte le contenu).

**Communication entre les deux parties**

1. Le **frontend** (React) envoie une requête HTTP à l’API (GET, POST, PUT, DELETE).
2. Le **backend** (Spring Boot) reçoit la requête dans le **Controller**, la traite via le **Service**, et accède à la base via le **Repository**.
3. Le backend renvoie une **réponse JSON** que React affiche à l’écran.