Yes. Below is a **clean, production-appropriate PostgreSQL version** of the same **Spring Boot + React Employee Management System**, with only the **necessary and correct changes**. The frontend **does not change at all**—only the backend configuration and dependencies do.

I will structure this so you can **directly apply it** without confusion.

---

1. Backend Changes — PostgreSQL

1.1 Maven Dependency (IMPORTANT)

Remove **MySQL** dependency and add **PostgreSQL**.

pom.xml

```
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
</dependency>
```

You should already have:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

No other database dependencies are required.

1.2 PostgreSQL Database Setup

Create database manually:

```
CREATE DATABASE emsdb;
```

(Optional but recommended)

```
CREATE USER emsuser WITH PASSWORD 'ems123';
GRANT ALL PRIVILEGES ON DATABASE emsdb TO emsuser;
```

1.3 `application.properties` (PostgreSQL Version)

Replace your existing datasource config with this:

```
# PostgreSQL Configuration
spring.datasource.url=jdbc:postgresql://localhost:5432/emsdb
spring.datasource.username=emsuser
spring.datasource.password=ems123

# JPA / Hibernate
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true

# Server
server.port=8080
```

If PostgreSQL is running on default port **5432**, no change is needed.

1.4 Entity (NO CHANGE REQUIRED)

Your entity class already works correctly with PostgreSQL.

`Employee.java`

```
@Entity
@Table(name = "employees")
public class Employee {

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

    private String firstName;
    private String lastName;

    @Column(unique = true)
    private String email;
}
```

Note: `GenerationType.IDENTITY` works correctly with PostgreSQL.

1.5 Repository (NO CHANGE)

```
public interface EmployeeRepository extends JpaRepository<Employee, Long> {
}
```

1.6 Controller (NO CHANGE)

Your REST controller remains exactly the same.

---

2. Frontend — React (NO CHANGES)

PostgreSQL is a **backend concern only**.

Your React code:

- Axios
- Routing
- Components

remain **100% unchanged**.

The frontend still calls:

```
http://localhost:8080/api/employees
```

---

3. Run Order (IMPORTANT)

Step 1: Start PostgreSQL

Make sure PostgreSQL service is running.

```
psql -U emsuser -d emsdb
```

Step 2: Run Backend

```
mvn spring-boot:run
```

Hibernate will automatically create the `employees` table.

Step 3: Run Frontend

```
npm start
```

---

4. PostgreSQL Table Auto-Generated by Hibernate

Hibernate will create something equivalent to:

```
CREATE TABLE employees (
    id BIGSERIAL PRIMARY KEY,
    first_name VARCHAR(255),
    last_name VARCHAR(255),
    email VARCHAR(255) UNIQUE
);
```

---

5. Common PostgreSQL Issues (And Fixes)

❌ Error: `relation "employees" does not exist`

✔ Fix:

```
spring.jpa.hibernate.ddl-auto=update
```

❌ Error: `FATAL: password authentication failed`

✔ Fix:

- Check username/password
- Check `pg_hba.conf`
- Restart PostgreSQL

---

6. Final Architecture (Clean)

```
React (Axios)
   |
   | HTTP JSON
   v
Spring Boot REST API
   |
   | JPA / Hibernate
   v
PostgreSQL Database
```

---

7. Professional Recommendation

If your goal is **industry-ready learning**:

- PostgreSQL > MySQL
- Spring Boot + React + PostgreSQL is a **standard enterprise stack**
- This project is **resume-worthy** if documented properly

If you want, I can next:

1. Add **DTO + Exception Handling**
2. Add **Pagination & Sorting**
3. Add **Docker (Postgres + Spring Boot)**
4. Convert this into a **proper GitHub project structure**

Tell me how far you want to take it.