Spring JDBC connectivity using spring boot and h2 console and using command line runner

📌 Refined and Polished Notes on Spring Boot JDBC with `CommandLineRunner`

1️⃣ Repository Class (`PersonJdbcDao`) - Database Interaction

- The **repository class** is responsible for performing **database operations**.
- It uses `JdbcTemplate`, which eliminates the need for writing **boilerplate code** for database connection, query execution, and resource management.
- `JdbcTemplate` **automates the process** of setting up and managing database interactions.

2️⃣ Entity Representation (`Person` Class) - Mapping Database Rows to Objects

- The repository fetches **raw data from the database**, but Java **does not inherently understand** SQL table rows.
- To bridge this gap, we create the `Person` **class** (not an actual `@Entity` since we are using JDBC, not JPA).
- The **responsibility of the** `Person` **class** is to **map database rows to Java objects**, making the data usable in our application.

3️⃣ Using `BeanPropertyRowMapper` for Automatic Mapping

- `BeanPropertyRowMapper<Person>` is used to **convert database records** into `Person` objects.
- It automatically maps **column names** in the database table to the **corresponding fields** in the `Person` class.

- **Important:** The `Person` class **must include a no-argument constructor**, as `BeanPropertyRowMapper` relies on it to instantiate objects.

4️⃣ Why Do We Need the `Person` Class?

- The purpose of the `Person` class is **twofold**:
  1. **To return data to the frontend** in a structured format (in real-world applications).
  2. **To verify that database operations are working correctly** by checking the results in the command line interface (CLI), without needing to manually log in to the database.

5️⃣ Displaying Results Without Database Login - Using Logger and `CommandLineRunner`

- **Instead of manually checking database records**, we print the results **directly to the CLI** using a **logger (`slf4j`)**.
- `CommandLineRunner` provides a **method (`run()`) that executes automatically** after the application starts, allowing us to fetch and display the data.

6️⃣ Logging the Output to CLI

- **To log database results, we use SLF4J (`LoggerFactory`).**
- A **logger instance is created** to log the fetched data:

```
java
CopyEdit
private static final Logger logger = LoggerFactory.getLogg
er(DatabaseDemoApplication.class);
```

- **Why do we pass `DatabaseDemoApplication.class` to `LoggerFactory`?**
  - It ensures that logs are categorized under the **main application class**, making debugging easier.

7️⃣ Injecting the Repository (`@Autowired`) and Fetching Data

- The **repository class (`PersonJdbcDao`) is injected** using `@Autowired`, allowing us to access its methods.
- The `run()` method (from `CommandLineRunner`) **automatically invokes `findAll()`**, retrieving the data from the database.
- **Since `findAll()` returns a list of `Person` objects,** Java calls the `toString()` method automatically when logging.

8️⃣ Execution Flow of the Application

1️⃣ Spring Boot starts and initializes all components (`PersonJdbcDao`, `Person`, `Logger`).

2️⃣ Since `DatabaseDemoApplication` implements `CommandLineRunner`, the `run()` method executes automatically.

3️⃣ `run()` calls `dao.findAll()`, fetching all records from the database.

4️⃣ Each database row is mapped to a `Person` object using `BeanPropertyRowMapper`.

5️⃣ The `Logger` prints the fetched data to the console.

6️⃣ The `toString()` method ensures that the log output is formatted neatly.

📌 Final Summary

✅ **`PersonJdbcDao` (Repository) interacts with the database using `JdbcTemplate`.**

✅ **The `Person` class converts database rows into Java objects.**

✅ **`BeanPropertyRowMapper` automatically maps the columns to fields in `Person`.**

✅ **`CommandLineRunner` executes `findAll()` automatically after startup.**

✅ **The `Logger` logs the results in the CLI to avoid manually querying the database.**

✅ **The overridden `toString()` method ensures that data is displayed in a clean format.**

🚀 Conclusion

This setup allows us to **fetch, map, and display database records automatically** when the application starts, without needing to manually check the database.

Would you like to extend this by adding **Insert, Update, or Delete operations**? 🚀

📌 Refined and Polished Notes on Spring Boot JDBC with `CommandLineRunner`

1️⃣ Repository Class (`PersonJdbcDao`) - Database Interaction

- The **repository class** is responsible for performing **database operations**.
- It uses `JdbcTemplate`, which eliminates the need for writing **boilerplate code** for database connection, query execution, and resource management.
- `JdbcTemplate` **automates the process** of setting up and managing database interactions.

2️⃣ Entity Representation (`Person` Class) - Mapping Database Rows to Objects

- The repository fetches **raw data from the database**, but Java **does not inherently understand** SQL table rows.
- To bridge this gap, we create the `Person` **class** (not an actual `@Entity` since we are using JDBC, not JPA).
- The **responsibility of the** `Person` **class** is to **map database rows to Java objects**, making the data usable in our application.

3️⃣ Using `BeanPropertyRowMapper` for Automatic Mapping

- `BeanPropertyRowMapper<Person>` is used to **convert database records** into `Person` objects.
- It automatically maps **column names** in the database table to the **corresponding fields** in the `Person` class.
- **Important:** The `Person` class **must include a no-argument constructor**, as `BeanPropertyRowMapper` relies on it to instantiate objects.

4️⃣ Why Do We Need the `Person` Class?

- The purpose of the `Person` class is **twofold**:
  1. **To return data to the frontend** in a structured format (in real-world applications).
  2. **To verify that database operations are working correctly** by checking the results in the command line interface (CLI), without needing to manually log in to the database.

5️⃣ Displaying Results Without Database Login - Using Logger and `CommandLineRunner`

- **Instead of manually checking database records**, we print the results **directly to the CLI** using a **logger (`slf4j`)**.
- `CommandLineRunner` provides a **method (`run()`) that executes automatically** after the application starts, allowing us to fetch and display the data.

6️⃣ Logging the Output to CLI

- **To log database results, we use SLF4J (`LoggerFactory`).**
- A **logger instance is created** to log the fetched data:

```
java
CopyEdit
private static final Logger logger = LoggerFactory.getLogg
er(DatabaseDemoApplication.class);
```

- **Why do we pass `DatabaseDemoApplication.class` to `LoggerFactory`?**
  - It ensures that logs are categorized under the **main application class**, making debugging easier.

7️⃣ Injecting the Repository (`@Autowired`) and Fetching Data

- The **repository class (`PersonJdbcDao`) is injected** using `@Autowired`, allowing us to access its methods.
- The `run()` method (from `CommandLineRunner`) **automatically invokes `findAll()`**, retrieving the data from the database.
- **Since `findAll()` returns a list of `Person` objects,** Java calls the `toString()` method automatically when logging.

8️⃣ Execution Flow of the Application

1️⃣ Spring Boot starts and initializes all components (`PersonJdbcDao`, `Person`, `Logger`).

2️⃣ Since `DatabaseDemoApplication` implements `CommandLineRunner`, the `run()` method executes automatically.

3️⃣ `run()` calls `dao.findAll()`, fetching all records from the database.

4️⃣ Each database row is mapped to a `Person` object using `BeanPropertyRowMapper`.

5️⃣ The `Logger` prints the fetched data to the console.

6️⃣ The `toString()` method ensures that the log output is formatted neatly.

📌 Final Summary

✅ **`PersonJdbcDao` (Repository) interacts with the database using `JdbcTemplate`.**

✅ **The `Person` class converts database rows into Java objects.**

✅ **`BeanPropertyRowMapper` automatically maps the columns to fields in `Person`.**

✅ **`CommandLineRunner` executes `findAll()` automatically after startup.**

✅ **The `Logger` logs the results in the CLI to avoid manually querying the database.**

✅ **The overridden `toString()` method ensures that data is displayed in a clean format.**

🚀 Conclusion

This setup allows us to **fetch, map, and display database records automatically** when the application starts, without needing to manually check the database.

Would you like to extend this by adding **Insert, Update, or Delete operations**? 🚀