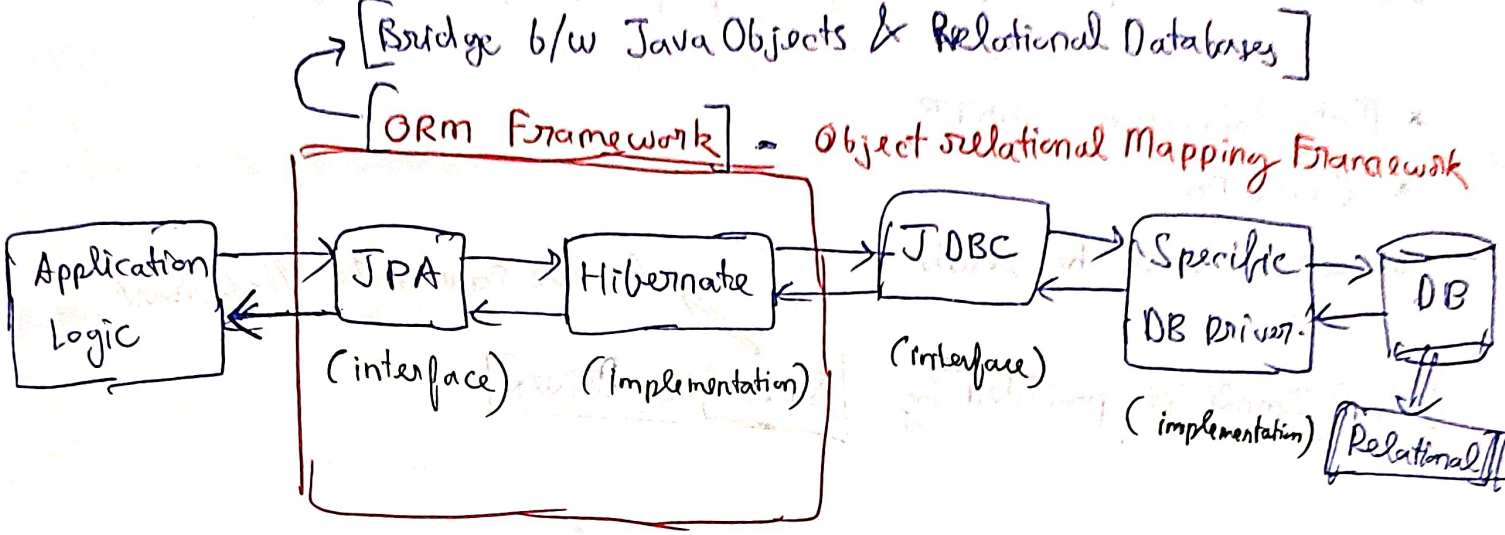
Video 22: Spring Boot: JPA (Part 1) | JDBC template



Till Now, we have only seen changes and Spring Boot related knowledge till Application Logic only.

Lets see how Application interacts with Database.

JPA → Its an interface, it only exposes us APIs

Hibernate / OpenJPA → These provide us the implementation of the APIs exposed to us. to interact with DB.

Hibernate internally talks with JDBC

Specific Drivers:
Each set of Drivers have their own set of implementations

JDBC → Java Database Connectivity provides an interface to

* Make Connection with DB
* Query DB
* and process the result

As we saw in the flowchart.

Actual Impl is provided by [Specific Drivers]

MySQL:

* Driver: Connector/J
  - Class: com.mysql.cj.jdbc.Driver

PostgreSQL

* Driver: PostgreSQL JDBC driver
* Class: org.postgresql.Driver

H2 (in-memory)

* Driver : H2 Database Engine
* Class : org.h2.Driver

So if our dB changes, we change the Specific Driver, just that. Implementation, we don't need to do anything

Now Lets see problems of Using JDBC without Spring Boot.

```
public class DatabaseConnection{

    public Connection getConnection(){

        try{
            // h2 driver loading
            Class.forName("org.h2.Driver");
            // Establish DB connection
            return DriverManager.getConnection("jdbc:h2:mem:userDB", "sa", "");
        }
        catch(ClassNotFoundException | SQLException e){
            // handle Exception
        }
        return null;
    }
}
```

Without Springboot we have to write DAO → Data Access Object.

In short Lot of code has to be written.

```
public class UserDAO{

    public void CreateUserTable(){

        try{
            Connection connection = new DatabaseConnection().getConnection();
            Statement statementQry = ------>
            String sql = "-----------";
            StatementQuery.executeUpdate(sql, statementQry);
        }
        catch(SQLException e){
        }
    }
}
```

Problem with Plain JDBC

- Connection
- Statement
- Prepared Statement
- ResultSet

→ JDBC interface provides interface for all of these. and each Specific driver provides implementation for this.

But there is so much BoilerPlate Code like:

- Driver Class Loading
- DB connection Making
- Exclusively Exception Handling
- Closing of the DB connections
- Manual Handling of DB Connection Pool.

↳ Using SpringBoot with JDBC helps us to remove all this boiler plate code for us.

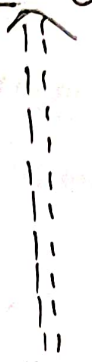
[Using JDBC with SpringBoot]

pom.xml.

dependency required to add in pom.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

Spring boot provides JdbcTemplate class, which helps to remove all the boiler code.



```
@Repository
public class UserRepository{

    @Autowired
    JdbcTemplate jdbcTemplate;

    public void createTable(){
        jdbcTemplate.execute("------");
            // create table execute query.
    }
    public void insertUser(String name, int age){
        String insertQuery ="-------";
          //query to insert
        jdbcTemplate.update(insertQuery, name, age);
    }
    public List<User> getUsers(){
        String selectQuery ="-------";
        return jdbcTemplate.query("------");
    }
}
```

POJO

```
public class User{
    int userId;
    String userName;
    int age;
}
```

As you can see JDBCTemplate has reduced the code so much,

We don't need to take care of making connection, handling Exception, Closing Connection.

application.properties

```
spring.datasource.url = jdbc:h2:mem:userDB
spring.datasource.driver-class-name = org.h2.Driver
spring.datasource.username = sa
spring.datasource.password =
spring.h2.console.enabled = true
```

DataSource → Database Connection

DataSource → ConnectionPool (HikariConnectionPool)

In plain JDBC if you remember, we created a DatabaseConnection Class

```
public class DatabaseConnection {
    public Connection getConnection() {
        try {
            Class.forName("jdbc:h2:mem:userDB");
            ;
        }
    }
}
```

(Plain JDBC)

Now these properties will get picked up from application.properties and SpringBoot will internally take care of Creating the Connection

Advantages

Driver Loading : JDBC Template load at the time of ~~app~~ application Startup in [DriverManager.class]

When you start the application, there is a class called.

(DriverManager.java)

```
isDriverAllowed ( ) {

    Class.forName( )

}
```

⇒ the code which we wrote manually ~~do~~ to get a Database connection.
SpringBoot is directly doing it for us.

DB Connection Making :- JDBCTemplate takes care of it, whenever we execute any query

[jdbcTemplate.update( ), jdbcTemplate.query( )]

during these method calls jdbcTemplate tries to create connection. or it will take from the pool.

Exception Handling :- In plain JDBC, we get very abstracted 'SQLException' but in jdbcTemplate, we get granular level Exception.

("DuplicateKeyException", "QueryTimeOutException").

Closing of the DB connection and other resources.

→ When we invoke update or query method, after success or failure of the operation, JDBCtemplate takes care of either closing or return the connection to Pool itself.

Manual Handling of DB Connection Pool. :—

In the application properties, we have not provided which DataSource to use. So by default it uses the default jdbc connection pool i.e. "HikariCP" with min and max pool configuration.

```
spring.datasource.hikari.maximum-pool-size=10
spring.datasource.hikari.minimum-idle=5
```

Configs to provide in application.properties

We can customise our data source and use something else as well

```
@Configuration
public class AppConfig {

    @Bean
    public DataSource dataSource() {
        HikariDataSource dataSource = new HikariDataSource();
        ....
        return dataSource;
    }
}
```

We can use any other dataSource, if we want

Many helper methods are there for JDBCTemplate