CPAN-252 WEB APPLICATION DEVELOPMENT FOR CPAN DIPLOMA. PROF. ANTON KOVUNOV

WORKING WITH DATA IN SPRING USING DATABASE AND TOOLS TO HELP WITH IT

READING AND WRITING DATA WITH JDBC

- Historically working with relational databases and SQL was the leading choice of data persistence. Plain Java has a sql package, which is sufficient for database work, but sometimes is too finicky in terms of setup. Spring helps us by providing two abstractions of JDBC and JPA.
- Let’s dive into things, first we will see more familiar to us - JDBCTemplate, then we will move to higher level abstraction

WHAT IS JDBC (REFRESHER)

- JDBC (Java Database Connectivity) is the Java API that manages connecting to a database, issuing queries and commands, and handling result sets obtained from the database. Released as part of JDK 1.1 in 1997, JDBC was one of the earliest libraries developed for the Java language.
- JDBC was initially conceived as a client-side API, enabling a Java client to interact with a data source. That changed with JDBC 2.0, which included an optional package supporting server-side JDBC connections. Every new JDBC release since then has featured updates to both the client-side package (java.sql) and the server-side package (javax.sql). JDBC 4.3, the most current version as of this writing, was released as part of Java SE 9 in September 2017 as JSR 221.

IMPLEMENT SIMPLE FIND BY ID OPERATION USING JAVA, LOOKS NO GOOD :)

```
@Override
public Optional<Ingredient> findById(String id) {
  Connection connection = null;
  PreparedStatement statement = null;
  ResultSet resultSet = null;
  try {
    connection = dataSource.getConnection();
    statement = connection.prepareStatement(
        "select id, name, type from Ingredient where id=?");
    statement.setString(1, id);
    resultSet = statement.executeQuery();
    Ingredient ingredient = null;
    if(resultSet.next()) {
      ingredient = new Ingredient(
          resultSet.getString("id"),
          resultSet.getString("name"),
          Ingredient.Type.valueOf(resultSet.getString("type")));
    }
    return Optional.of(ingredient);
  } catch (SQLException e) {
    // ??? What should be done here ???
  } finally {
    if (resultSet != null) {
      try {
        resultSet.close();
      } catch (SQLException e) {}
    }
    if (statement != null) {
      try {
        statement.close();
      } catch (SQLException e) {}
    }
    if (connection != null) {
      try {
        connection.close();
      } catch (SQLException e) {}
    }
  }
  return Optional.empty();
}
```

LET'S CONVERT IT TO OUT CASE BY USING JDBCTEMPLATE

- First of all, let's enrich our app to support database, we will work with H2 SQL in-memory DB as it's really lightweight, convenient for learning purposes and has great integration with Spring Devtools
- To read more specs about it, you can go to this page: https://www.h2database.com/html/main.html
- We also need to provide jdbc template support, both these operation are done through pom.xml file by adding dependencies

DEPENDENCIES DECLARATION

```
<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>
```

SETTING UP DATASOURCE

- By default when we start the app database will be spun up and DB name will be generated. We want to pin database name so we could easier track it using devtools.
- This is done by going to application.properties and providing some db configuration, which is self explanatory. Now the database url will be “jdbc:h2:mem:tekkenreborn”

```
spring.datasource.generate-unique-name=false
spring.datasource.name=tekkenreborn
```

TWEAKING FIGHTER MODEL TO BE INTEGRATED WITH DATABASE

- We will need to change our models of Fighter and CharacterPool to have proper fields for db integration, namely every model should have ID provided and timestamp field.

```java
@Data
public class Fighter {
    private Long id;
    private Date createdAt = new Date();
    private final String name;
    private final int damagePerHit;
    private final int health;
    private final double resistance;
    private final Anime animeFrom;
}
```

```java
@Data
public class CharacterPool {
    private Long id;
    private List<Fighter> fighters;
    public void addHero(Fighter fighter) {
        fighters.add(fighter);
    }
}
```

IMPLEMENTING REPOSITORY TO WORK WITH JDBC

- As we set up dependencies to support our data persistence and we adjusted models to align with database requirements, we need to specify service that will be responsible for interaction with the database
- In Spring we has specific type of Bean, called @Repository, which is responsible for implementation of database methods
- Let’s write a simple implementation together.

FIGHTER REPOSITORY INTERFACE

- First of all we will define interface with operations we want to be able to do with our Fighter table. Three methods will be supported, findAll(), findById() and save()

```java
package com.cpan252.tekkenreborn.data;

import java.util.Optional;

import com.cpan252.tekkenreborn.model.Fighter;

public interface FighterRepository {
    Iterable<Fighter> findAll();
    Optional<Fighter> findById(Long id);
    Fighter save(Fighter fighter);
}
```

FIGHTER JDBC REPOSITORY IMPLEMENTATION

- Let’s create an implementation without any methods, but with embedded JdbcTemplate.

- I skipped implementation for brevity, but you should also implement all the interface methods, as of now with null.

```
@Repository
public class JdbcFighterRepository implements FighterRepository {
    private JdbcTemplate jdbcTemplate;

    @Autowired
    public JdbcFighterRepository(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }
```

QUESTIONS?

On the lab I am going to explain autowired annotation, how beans work in Spring overall, and we will provide database implementation and do our first assignment.