INSTRUCTIONS

Spring Data JPA

Purpose

In this lab you will gain experience with Spring Data JPA. You will see that Spring Boot greatly simplifies handling data through JPA without the plumbing necessary with Spring or standalone apps.

Learning Outcomes

What you will learn:

1. How to implement a Spring JPA application using Spring Boot.
2. How to create Spring Data repositories using JPA.

Specific subjects you will gain experience with:

1. Spring Data JPA
2. Hibernate ORM

You will be using the *34-spring-data-jpa* project.

Estimated time to complete: 30 minutes.

Use Case

In the previous lab you "bootified" a Spring application, showing how to simplify it and wrap it with a runtime.

In this lab you will take it further by replacing Account and Restaurant JDBC repositories with use of Spring Data JPA.

The Rewards repository will remain JDBC, which we will review why later in the lab.

The scope of the lab is *not* to refactor from JDBC. Refactoring between JDBC and JPA data access is beyond the scope of this course.

The starting point of this lab will include pre-annotated data entity classes, and empty repository references that you will implement JPA backing entities, and accompanying repositories to demonstrate the ease of implementation.

Quick Instructions

If you are already knowledgeable with the lesson concepts, you may consider jumping right to the code, and execute the lab in form of embedded TODO comments. Instructions on how to view them are here.

If you aren't sure, try the TODO instructions first and refer to the lab instructions by TODO number if you need more help.

Instructions

Declare JPA Dependencies

TODO-01 : Check JPA dependencies for the project.

- Review the JPA Starter to `pom.xml` or `build.gradle`. Take a moment to locate and review that the `spring-boot-starter-data-jpa` dependency has already been configured for you.

  What does this dependency do?

  Adding Spring Boot Data JPA starter will add a tremendous amount of behavior to your project, some of the major functionality being:

  - `Repository` interface

    Spring Data Commons gives functionality to declaratively define an interface for a repository without requiring an explicit implementation class - Spring Data will implement the behavior via a Spring proxy. See Spring Data Commons - Repositories for more information.

  - JPA

  - Hibernate

  - JDBC

  - Transactions

  - AOP (given that it uses aspects behind the scenes to wire transaction managers and other cross cutting behavior to your Spring proxies).
- Given JPA includes JDBC by default, you may remove the JDBC Starter from your `pom.xml` or `build.gradle`. Or, you may leave it to explicitly declare the dependency given you will also use `JdbcTemplate` in your code.

Implement JPA for `Account`

TODO-02 : Review the JPA annotations on the *Account* class and make sure you know what each does.

- `@Entity` - Marks class as a JPA persistent class.
- `@Table` - Specifies the exact table name to use on the DB (would be "Account" if unspecified).
- `@Id` - Indicates the field to use as the primary key on the database.
- `@Column` - Identifies column-level customization, such as the exact name of the column on the table.
- `@OneToMany` - Identifies the field on the 'one' side of a one to many relationship.
- `@JoinColumn` - Identifies the column on the 'many' table containing the column to be used when joining. Usually a foreign key.

What might be a concern when annotating your domain classes with JPA behavior?

- You are tying JPA and ORM to your domain entities.

- You are requiring JPA and ORM coupled with your runtime.

In reasonably complex projects you may want to keep JPA coupled code independent of POJO domain objects, and use either *Proxies* or *Adapters* to decouple them.

TODO-03 : Implement the `AccountRepository` to be a Spring Data JPA Repository.

- Make the `AccountRepository` to be a Spring Data JPA Repository interface.

- Use a renaming refactoring to change the `findByCreditCard` method to obey Spring Data conventions.

  Why do you need to do this?

  Spring Data JPA requires specific pattern of naming of your methods to determine how to construct the associated data access logic.

TODO-04 : Review the JPA annotations on the *Beneficiary* class and make sure you know what each does.

- `@AttributeOverride` - Tells JPA to use the `ALLOCATION_PERCENTAGE` column on `T_ACCOUNT_BENEFICIARY` to populate `Percentage.value`.

Implement JPA for `Restaurant`

TODO-05 : Map the `Restaurant` class using JPA annotations.

You thought you would get away without having to annotate an entity class? Think again!

In this step you will map the `Restaurant` entity to a database structure that is already defined.

- Annotate the `Restaurant` class as a JPA Entity.
- Use the `schema.sql` file to provide mapping.
  - Map the table.
  - Map the columns.
- You need to map `Percentage.value` from a column in `T_RESTAURANT`. Percentage is not a simple type, so you will have to override it.

  Refer to the following annotation to the `benefitPercentage` member if you need help:

  ```
  @AttributeOverride(name="value",column=@Column(name="BENEFIT_PERCENTAGE"))
  ```

TODO-06 : Implement the `RestaurantRepository` to be a Spring Data JPA Repository.

- Make the `RestaurantRepository` to be a Spring Data JPA Repository interface.
- Use a renaming refactoring to change the `findByMerchantNumber` method to obey Spring Data conventions.

Configure for JPA

As you learned from the lecture material, if you were building a basic Spring JPA project you would also need to configure your project to enable the use of JPA and Spring Data Repositories by performing the following tasks.

- Enable wiring of previously defined JPA annotations to an ORM backend through an `EntityManager`.
- Enable use of Hibernate as the ORM backend. Hibernate would be wired to the datasource that is auto-configured.
- Enabled scanning for JPA repositories using the `@EnableJpaRepositories` annotation.

- Set up a `TransactionManager` and enable transactions with `@EnableTransactionManagement`.

However, since this is a Spring Boot enabled test and Autoconfiguration is enabled, you will find that it is not necessary to do any of these. This allows you to focus on configuring just the custom values needed to override default configurations.

TODO-07 : Configure JPA

The previous steps cover a set of reasonable defaults for wiring together an application to use JPA and ORM.

But what if we want to override or extend behavior?

You will demonstrate configuring JPA in `src/test/resources/application.properties` as follows:

- Run test SQL scripts (`test-schema.sql` and `test-data.sql`) located under `rewards.testdb` directory
- Show SQL execution
- Format SQL shown
- Bypass DDL execution on startup

The last option is a common configuration in production environments, where it is not desired to re-run DDL.

You may be wondering *why are there two application properties*?

There are two in your project at the following locations:

- `src/main/resources/application.properties`
- `src/test/resources/application.properties`

They are files for externalizing configuration from your code. You have separate classpaths for the main application, as well as the test.

Given you are setting a specific configuration set for testing is why you updated the one in the test classpath.

Run It

TODO-08 : Run the test, it should pass.

- Run `RewardNetworkTests`. Verify it succeeds.
- Review the console carefully for hibernate execution of SQL statements.

Summary

In this lab you implemented `Account` and `Restaurant` JPA repositories. You also showed some basic configuration for *Hibernate*.

You did not implement JPA for `Rewards` Repository. Why not?

If you look carefully at the `JdbcRewardsRepository.confirmReward()` method, you will notice the naming of the method is more of a business vocabulary than repository. It does not map to Spring Data conventions.

Could you port `RewardsRepository` to JPA? How might you do this?

Congratulations, you are done with the lab!