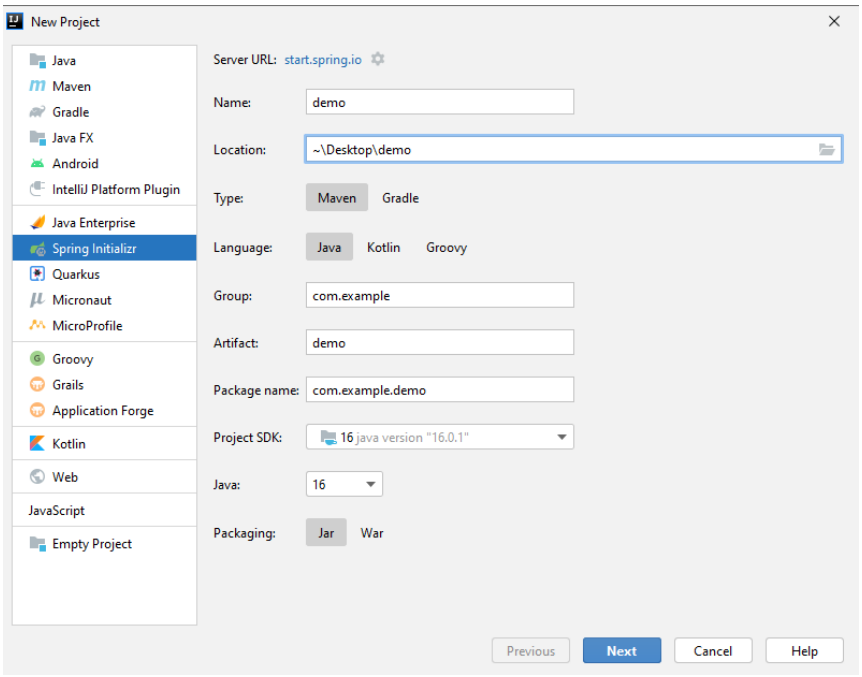
Lab: Spring Data – Account System

This document defines the lab assignments for the "Spring Data" course @ Software University.

Your task is to create a simple account system that has users with accounts and manages money transfer or withdrawal. Build the system using code-first and Spring Data. The goal is to implement services and repositories.

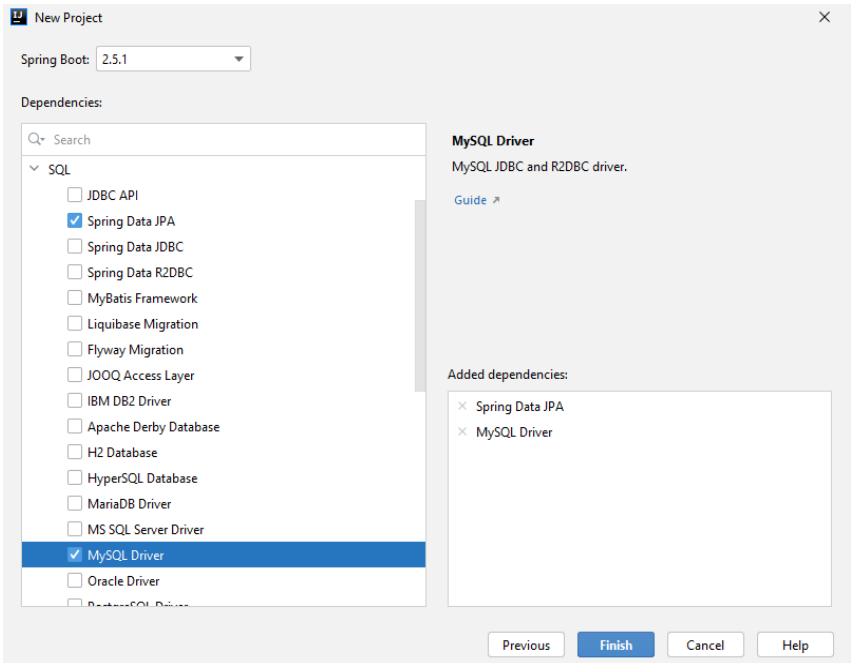
1. Project Setup

Create new Spring project:



Add name and version:

Add Spring Data JPA:



In the resources folder, add new **applications.properties** file, which will hold the Spring configuration of the project:

```
#Data Source Properties
spring.datasource.driverClassName = com.mysql.cj.jdbc.Driver
spring.datasource.url = jdbc:mysql://localhost:3306/new_lab?useSSL=false
spring.datasource.username = root
spring.datasource.password = 12345

#JPA Properties
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL8Dialect
spring.jpa.properties.hibernate.format_sql = TRUE
spring.jpa.hibernate.ddl-auto = create-drop
###Logging Levels
# Disable the default loggers
logging.level.org = WARN
logging.level.blog = WARN

#Show SQL executed with parameter bindings
logging.level.org.hibernate.SQL = DEBUG
logging.level.org.hibernate.type.descriptor = TRACE
```

Start splitting the java directory into packages. Create several ones to help you organize your project:

- **models** – the directory of our database models(entities)
- **repositories** – the package where we will hold the repository interfaces
- **services** – where our service interfaces and implementations will be stored

2. Database Models

Start by setting up the database models. Each one of them will be as follows:

- **User**
  - **Id** – long value**, primary key**
  - **Username** – **unique** for each user
  - **Age** – integer value
  - **Accounts** – each user can have **many accounts**, which will be **identified by their id**
- **Account**
  - **Id** – long value, **primary key**
  - **Balance** – BigDecimal
  - **User** – an account can be owned by a **single user**

Set up appropriate tables, columns, column properties and table relations.

3. Repositories

Spring Data reduces the amount of boiler-plate code by using a central interface **`Repository`**.
The **`JpaRepository`** interface contains methods like:

- **`save(E entity)`**
- **`findOne(Id primaryKey)`**
- **`findAll()`**
- **`count()`**
- **`delete(E entity)`**

- `exists(Id primaryKey)`

You can define a **custom repository**, which extends the **`JpaRepository`** and defines several methods for operating with data besides those exposed by the greater interface. The query builder mechanism of Spring Data requires following several rules when you define custom methods. Query creation is done by parsing method names by prefixes like **`find…By`**, **`read…By`**, **`query…By`**, **`count…By`**, and **`get…By`**. You can add more criteria by concatenating **`And`** and **`Or`** or apply ordering with **`OrderBy`** with sorting direction **`Asc`** or **`Desc`**.

Create two Repository interfaces – **`UserRepository`** and **`AccountRepository`**.

```
@Repository
public interface AccountRepository extends JpaRepository<Account, Long> {
    Account findAccountById(Long id);
}
```

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {

}
```

Add several methods to help you look up the data source, for example **`getByUsername(String username)`** in the **`UserRepository`** interface.

4. Services

In bigger applications mixing business logic and crud operations to the database is not wanted. Having a repository objects is implementing the **Domain Driven Design**. Repositories are classes responsible **only for write/transactional operations** towards the data source. Any business logic like validation, calculations and so on is implemented by **a Service Layer**. One of the most important concepts to keep in mind is that a **service** should **never expose details of the internal processes,** or the business entities used within the application.

Define several service **interfaces**:

```
public interface AccountService {
    void withdrawMoney(BigDecimal money, Long id);
    void transferMoney(BigDecimal money, Long id);
}
```

```
public interface UserService {
    void registerUser(User user);
}
```

Implement those services with classes **`AccountServiceImpl`** and **`UserServiceImpl`**. Those classes will do the business logic of the application. To do that, they should have certain type of **`Repository`** available – **`AccountRepository`** or **`UserRepository`** according to the service type.

```java
@Service
public class AccountServiceImpl implements AccountService {

    private final AccountRepository accountRepository;

    @Autowired
    public AccountServiceImpl(AccountRepository accountRepository) {
        this.accountRepository = accountRepository;
    }

    public void withdrawMoney(BigDecimal money, Long id) {...}

    public void transferMoney(BigDecimal money, Long id) {...}
}
```

```java
@Service
public class UserServiceImpl implements UserService{

    private final UserRepository userRepository;

    @Autowired
    public UserServiceImpl(UserRepository userRepository) {...}

    public void registerUser(User user) {...}
}
```

In Spring Data Framework, the usage of **@Service**, **@Repository** or **@Component** annotations is needed to separate different "**layers**" in the application. They are mainly used for programmers to know a class's role and which logical layer it belongs to.

The **@Autowired** annotation is required when **injecting a resource**, e.g., **Repository** to **Service**.

The implementation of the methods is up to you. Here are some several tips:

- **AccountServiceImpl**
  - Money withdrawal – should only happen if account **is present** in the database, **belongs to user** and **has enough balance**
  - Money transfer – should only happen if **account belongs to user** and transfer value **is not negative**
- **UserServiceImpl**
  - User registration – should only happen if user does not exist in the database

5. ConsoleRunner and Application

We will test our application in a ConsoleRunner class. Create such and inject needed repositories:

```
@Component
public class ConsoleRunner implements CommandLineRunner {

    private UserService userService;
    private AccountService accountService;

    @Autowired
    public ConsoleRunner(UserService userService, AccountService accountService){
        this.userService = userService;
        this.accountService = accountService;
    }

    @Override
    public void run(String... args) throws Exception {}
}
```

6. Test

Test the application by adding some logic in the **`ConsoleRunner`** class's method **`run`**:

```
@Override
public void run(String... args) throws Exception {
    User user = new User( username: "Pesho",  age: 20);

    Account account = new Account(new BigDecimal( val: "25000"));
    account.setUser(user);

    user.setAccounts(new HashSet<>() {{
        add(account);
    }});

    userService.registerUser(user);

    accountService.withdrawMoney(new BigDecimal( val: "20000"), account.getId());
    accountService.transferMoney(new BigDecimal( val: "30000"), account.getId());
}
```

If you have written everything correctly, an **account_system** database should be created with tables:

- **users**
- **accounts**
- **users_accounts**