Birzeit University
Faculty of Engineering and Technology
Electrical and Computer Engineering Department
Advance Computer Systems Engineering Lab ENCS515

EXP. No. 10. Spring Boot Part 2

1. Requirements

- ❖ Knowledge of the java programming language
- ❖ Basic Understanding of MVC (Model View Controller) architecture
- ❖ Basic Knowledge of Maven tool
- ❖ Basic Idea about Spring framework (Preferably Spring MVC)
- ❖ Basic Knowledge of relational databases (like MySQL)
- ❖ Knowledge of java ORM (Object Role Modeling) models is extremely helpful but not required
- ❖ Spring Tool Suite software (STS) (For windows 64-bit download this Link)
- ❖ Postman software (For windows 64-bit download this Link)
- ❖ MySQL server/Workbench (For windows 64-bit download this Link)

2. Objectives

- ❖ Understanding the fundamentals of Spring Data JPA framework
- ❖ Create mapping between java models and relational database models
- ❖ Connect and perform CRUD operations to MySQL database using Spring Data JPA framework

3. Introduction

- ORM

ORM (Object relational mapping) is a technique to convert data between models within object-oriented languages to serialized data to be stored (In relational database in our case). ORM models presents many advantages, the main advantage is the decoupling to the database connector, databases queries vary between one another, so ORM takes care of this difference.

- JPA

JPA (Java Persistence API) is a specification or standard which provides java developers with ORM to manage relational database (map entity classes to sql tables), after configuring the mapping it’s sent to some framework to handle it.

- Spring Data JPA

Spring Data JPA will be the framework to handle the mapping as we mentioned, it basically helps implementing repositories to retrieve and manipulate data in database. Spring Data JPA is built on top of Hibernate, hibernate is a very commonly used framework for ORM in java.

4. Procedure

Go through EXP. No. 9 Spring Boot Part 1 experiment section 4.1 and section 4.2 above and create a new project, required dependences: Spring web, Spring Data JPA, MySQL Driver.

4.1. Adding Dependencies: [This section is not required if you did that from the UI]

To run this application, we will use new dependencies as the JPA dependence and the MySQL dependency. These dependencies have some functions which will help us reduce the code and simplifies it.

➢ JPA Dependency

As we saw for Spring MVC Spring Boot gives us a meta dependency which is wrapper for all the dependencies we need for Spring Data JPA, to add the dependency go to pom.xml file and add the following dependency. This dependency contains all you need to create the needed repositories.

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

➢ MySQL Dependency

This dependency will help us connecting and accessing the data base to add this dependency go to pom.xml file and add the following dependency.

```
<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
</dependency>
```

After adding the dependencies, the pom.xml file will be as shown in the following code.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
	<modelVersion>4.0.0</modelVersion>
	<parent>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-parent</artifactId>
		<version>2.0.3.RELEASE</version>
		<relativePath/> <!-- lookup parent from repository -->
	</parent>
	<groupId>com.example</groupId>
	<artifactId>DataBase</artifactId>
	<version>0.0.1-SNAPSHOT</version>
	<name>DataBase</name>
	<description>Demo project for Spring Boot</description>

	<properties>
		<java.version>1.8</java.version>
	</properties>

	<dependencies>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-web</artifactId>
		</dependency>

		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-test</artifactId>
			<scope>test</scope>
		</dependency>

		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-data-jpa</artifactId>
		</dependency>

		<dependency>
			<groupId>mysql</groupId>
			<artifactId>mysql-connector-java</artifactId>
		</dependency>

	</dependencies>

	<build>
		<plugins>
			<plugin>
				<groupId>org.springframework.boot</groupId>
				<artifactId>spring-boot-maven-plugin</artifactId>
			</plugin>
		</plugins>
	</build>
	</project>
```

4.2. Configurations

We need some configuration for Hibernate (which JPA is built in top of) along with configuration to connect to database, keep in mind that spring Boot went far to provide an embedded database for development purposes, but this will not work for production purposes.

Configurations are set in src/main/resources/application.properties file we saw in the previous experiment, these configuration will be per database, which means configurations will change if the database changes and most likely this will be the only change, compared to JDBC templates this a big jump for database abstraction.

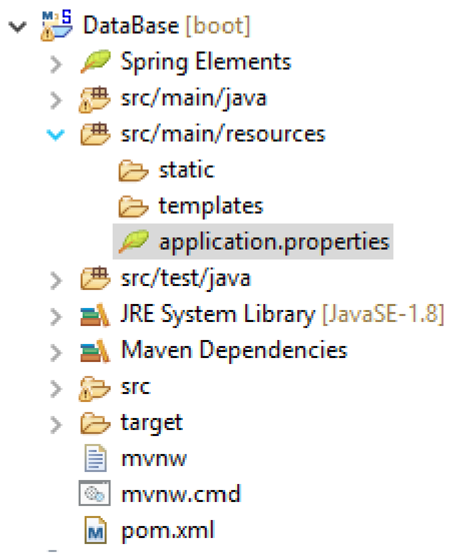

Figure 10.1 Application Properties

```
server.port=8081
spring.datasource.url=jdbc:mysql://localhost:3306/mydb?allowPublicKeyRetrieval=true&useSSL=false
spring.datasource.username=databaseUsername
spring.datasource.password=databaseUsernamePassword
spring.jpa.hibernate.ddl-auto=update
spring.jpa.hibernate.naming-strategy=org.hibernate.cfg.ImprovedNamingStrategy
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect
```

MyDB in the 2nd line is the database name, you need to fill the 3rd and 4th lines with your correct values. Following and on the next page, a description of them:

- **server.port:** This property overrides the default port Spring Boot will run on (8080).

- **spring.datasource.url:** This will be the URL for the database connection.

  useSSl=false: is used to disable SSL connection since database will be running on the same machine, in case database is running on different machine we need to configure certificate for both sides.
- **spring.datasource.username:** The username for the database (locally or in different machine)
- **spring.datasoruce.password:** The password for the database.
- **spring.jpa.hibernate.ddl-auto:** This option tells hibernate to recreate the database on each run, for production purposes we usually use update does not create.
- **spring.jpa.hibernate.naming-strategy:** This will define the default name strategy of the database tables and columns.
- **spring.jpa.properties.hibernate.dialect:** This defines the language which will be uses to talk to database.

4.3. ORM mapping:

The idea of JPA is to create mapping which makes it possible to convert java model to relational model, in this experiment we will take a quick look at the mapping using annotation based configuration, many enterprise application use xml based configuration but annotation is the new trend and most of the new projects are trying to eliminate the use of xml configurations. The structure of the project will be as in section 4.3. of Experiment 9. But, In this project we will add a new package called repositories in the com.example.demo package.

- Models and Tables:

  - Tables

In ORM classes are mapped to database table, this is achieved using **@Entity** annotation on the class definition.

- ♦ Id

Relational databases assign a primary key to each record in which it will be identified by, Spring data JPA uses the primary key to realize if a record needs to be added or updated, to specify a primary key we use @Id annotation, this annotation applies primary key to the attribute annotated, this applies not-null and unique constraints as well.

Primary keys are usually auto generated, there are many ways to generate them, here for simplicity we will be using auto increment for simplicity, we use @GeneratedValue(strategy = GenerationType.AUTO) , this strategy is by default set.

- ♦ Columns

Columns by default are scanned and persisted by JPA, we can use @Column annotation to apply more specific properties, we can specify the name of the column, the uniqueness and the ability to be null. To avoid persisting a column in the database table we can use @Transiet to tell JPA to ignore a specific attribute.

- ♦ User Class

In this walk through we will be implementing a User entity which has an Id and User Name, the Id is the main key which will be generated automatically, models in spring boot basically are normal classes, so in com.example.demo.models package go ahead and create a class User and add its attributes. Notice that all attributes should be private and should be accessed using accessors (getters and setters) to follow encapsulation standards.

```java
package com.example.demo.models;

import javax.persistence.Entity;
import javax.persistence.Table;
import javax.persistence.*;

@Entity
@Table(name = "users")
public class User {

	@Id
	@GeneratedValue(strategy = GenerationType.AUTO)
	@Column (name="user_id")
	private Integer id=null;

	@Column(nullable = false, unique = true)
	private String userName;

	public User(String userName) {
		super();
		// The ID is Auto-generated.
		this.userName = userName;
	}


	public User() {
		super();
		// TODO Auto-generated constructor stub
	}


	public Integer getId() {
		return id;
	}

	public void setId(Integer id) {
		this.id = id;
	}

	public String getUserName() {
		return userName;
	}

	public void setUserName(String userName) {
		this.userName = userName;
	}

}
```

➢ JPA repositories

JPA introduces interface-based models for accessing the database, implementation is not required, all we need is to create custom repositories interfaces and extend predefined interfaces provided by Spring data JPA to be able perform operations on database. will create a repository interface for user, this interface should extend a repository interface in JPA, we can extend CrudRepository or JpaRepository. JpaRepository provides additional functionality including pagination, yet if you are going to need only simple CRUD operations then CrudRepository is recommended to keep the separate of concerns.

Creating repository interface is simple, we only need to create an interface in the com.example.demo.repositories as shown in Figure 10.2 and extends one of the spring jpa repositories, for user we will create UserRepository as shown in the code below.

JpaRepository takes generic types, the first type is the entity which is User in our case, the second one is the type of the id, in our previous demonstration of the mapping relation we specified the type of the id to be Integer. JpaRepository requires the type to be serializable, for that we can use Long or Integer which are wrapper classes for the primitive types long and int.

Right-click on the package shown in Figure 10.2, select ‘New’ the ‘Interface’

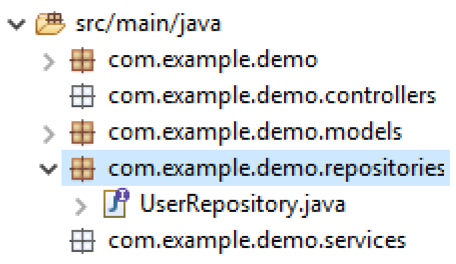
src/main/java
com.example.demo
com.example.demo.controllers
com.example.demo.models
com.example.demo.repositories
UserRepository.java
com.example.demo.services

Figure 10.2 User Repository

```java
package com.example.demo.repositories;

import org.springframework.data.jpa.repository.JpaRepository;

import com.example.demo.models.User;

public interface UserRepository extends JpaRepository<User, Integer> {

        }
```

- Service

Just that simple we have a DAO layer now, in the previous experiment we discussed that service layer should take care of controlling the DAO, so we need to inject our created DAO to the service. Spring JPA will take care of initializing the DAO, so we only need to wire it to our service, note that no need for any XML definition or annotation for the interface to be marked as a bean, extending the JPA/CRUD repositories is enough. You will create a UserService class in the com.example.demo.services package you created, and make an object of the repository you created, this interface is connected to the database which allows to perform queries on the database. Don’t forget to add the @autowired annotation before creating the repository object and @Service before the class definition to notify spring about service as shown in the following code.

```
package com.example.demo.services;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.example.demo.repositories.UserRepository;

@Service
public class UserService {

	@Autowired
	UserRepository userRepository;
}
```

Now we can add a new methods to get all users and add a user to the database using the UserRepository object by using findAll and save method respectively. Don’t forget to import the User class. These methods will be used by the controller which will be created next.

```
import com.example.demo.models.User;


	public List<User> getAll() {
		return userRepository.findAll();
	}
	public String addUser(User user) {
		userRepository.save(user);
		return "success";
	}
```

- Controller (APIs)

In a normal way controllers use services so we need a reference for our user service in the controller, on the startup of the application spring will initialize the singletone service bean (if you do not know what beans are please do some reading about spring beans) so we do not need to initialize service, we can wire the service to our controller using “@Autowired” annotation which wires the service bean to the reference defined in our controller.

That is all what we need to use UserService, so now we can use the user service to return all the user data by calling getUserList method we created in the UserService class.

```java
package com.example.demo.controllers;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.example.demo.models.User;
import com.example.demo.services.UserService;

@RestController
public class UserController {

	@Autowired
	UserService service;

	@RequestMapping("/users")
	public List<User> getAllUsers(){
		return service.getAll();
	}

	@PostMapping("/users")
	public String addOne(@RequestBody User user) {
		return service.addUser(user);
	}

}
```

Before running the application run the MySQL workbench and connect to the database, then you can run the application as we did in the previous experiment and test the methods you created.

➢ Delete

To delete object, we can use delete API which takes the id of the object to be deleted, we can write our own custom delete methods same as get methods.

```
public String deleteUser(Integer userId) {
        userRepository.deleteById(userId);
        return "success";
}
```

➢ Multiple Tables and relational database

When building applications, we are not writing data to single table, in our data model we often have relationship between entities. We will talk about different associations:

- One to One
- One to Many
- Many to Many

➢ One to One

one to one entity association is a relationship between two entities where the two side of relation maps to only one object in the other side. For this relation, we will use user/office relationship in which every user has an office and every office can be for only one user, we will create office class with some attributes.

```
package com.example.demo.models;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="office")
```

```
public class Office {
	@Id
	@GeneratedValue(strategy = GenerationType.AUTO)
	@Column (name="office_id")
	private Integer id=null;

	@Column (name="office_roomCode")
	private String roomCode;
}
```

To build the association we will create a field for user inside office, to map it to User entity we will user @OneToOne annotation, we will add cascade type persist, this mean when persisting office object , the user object added to it will be persisted as well, then we will add @JoinColumn which is the column that will be used to join the two entities.

```
	@OneToOne (cascade=CascadeType.PERSIST)
	@JoinColumn(name="user_id",referencedColumnName="user_id")
	private User user;
```

Name property in @JoinColumn will define the foreign key column in office table, referencedColumnName will define the primary key column in the target entity.

This is a unidirectional relation, we can access user from office, but we cannot access office from user, to make the relation bidirectional we can apply some changes.

First we need to add variable to hold the reference of office in user object, then we will annotate it with @OneToOne annotation, also we will give mappedBy = “user”, here we are telling jpa that this reference is mapped by user variable in office.

```
		@OneToOne (mappedBy="user")
		private Office office;
```

So now we have a bidirectional relationship where we can access office from user and user from office.

- One to Many

one to many is an association between two entities where one entity has reference to more than one entity while the other entity has reference to only one entity.

For this relation we will use user/account relation, one user can have many accounts while and account can belong to only one user. First, we will create account class with some attributes and accessors.

```
package com.example.demo.models;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="account")
public class Account {

	@Id
	@GeneratedValue(strategy = GenerationType.AUTO)
	@Column (name="account_id")
	private Integer id=null;

	@Column (name="account_codeNumber")
	private String codeNumber;

}
```

In User class we will add a collection of account, then we will use @OneToMany annotation to map the relation, then we will use @JoinColumn, this may be confusing because in one to one relationship this annotation was used in the owning entity (the entity which will have the foreign key) but in this case its clear that Account should have the foreign key which is the id of the user and not the opposite (User will have list of foreign keys since user can have list of accounts).

Currently this is a unidirectional relation (we will not have reference for user inside account object) so that leaves us with only the User to be able to specify the Join Column, JPA is smart enough to know that the Account is the owning entity.

Also, we will specify the cascade type to be persist so accounts get persisted along with the user, we will add additional attribute in the @JoinColumn which is nullable= false, this means each persisted account should be referenced by a user.

```
	@OneToMany(cascade = CascadeType.PERSIST)
	@JoinColumn(name = "user_id", nullable = false)
	private List<Account> accounts = new ArrayList<>();
```

Until now the relation is bidirectional, we can access accounts from user, but the opposite is not possible. To make the relation unidirectional first we will add a variable of type user inside Account class, now we will use @ManyToOne annotation, many accounts to one user , now we will specify the joincolumn using annotation @JoinColumn, note that we no longer need to specify the join column in user table so we will remove it, also we need to specify the variable user mapped by.

So, in User class we will have

```
@OneToMany(cascade = CascadeType.PERSIST, mappedBy="user")
@JoinColumn(name = "user_id", nullable = false)
private List<Account> accounts = new ArrayList<>();
```

In Account class, we will have

```
@ManyToOne
@JoinColumn(name="user_id")
private User user;
```

So now we have a bidirectional relation where we can access any side of the relation from the other side.

➢ Join Table:

Join table can be used as an alternative method to map one to many relationship, in many cases in one to many relationship, relationship could be not mandatory, association between two entities could occur but it’s not a must, in this case a junction table is used to avoid having null values in the foreign key columns where the relation is not there, junction tables are used mainly for many to many mapping but we will come to this later.

Junction table called also a join table, is a median table which holds the relationship between two tables, it will have a foreign key for both the tables and the 2 keys are the primary key of this table.

To demonstrate join tables, we will use relation user/car, note that every user can have many cars but a car can only belong to a one user, but a user also can have no car, to map this relation in jpa we will use @JoinTable annotation.

After creating car entity we will create a list of type car in user, we will put @OneToMany annotation since the relationship is one to many, the we will add @JoinTable, we need to specify the name of the junction table which will be user_car, then we will specify the join colum using attribute joinColumn for the user side and the inverseJoinColumns for the car side.

```
@OneToMany(cascade = CascadeType.PERSIST)
@JoinTable(name = "user_car", joinColumns=@JoinColumn(name="user_id"),
           inverseJoinColumns=@JoinColumn(name="car_id"))
private List<Account> cars = new ArrayList<>();
```

- Many to Many

many to many relationships are relationships where each entity can map to many entities in the other side of the relation, we will use user/course relationship, a user can be enrolled in many courses and course can have many users.

First, we will demonstrate a unidirectional relationship, we will have to pick the owning side of the relation, lets pick the user, so basically we should have a list of courses inside user, then we will use @ManyToMany annotation, we will also use @JoinTable annotation as we did in the previous section.

```
@ManyToMany(cascade = CascadeType.PERSIST)
@JoinTable(name = "user_course", joinColumns=@JoinColumn(name="user_id"),
             inverseJoinColumns=@JoinColumn(name="course_id"))
private List<Course> courses = new ArrayList<>();
```

So now we have a unidirectional relation where we can access courses from user, to make the relation bidirectional, we will not be changing anything on the user, in course entity we will establish an association.

In course entity we will create a list of users, again we will use @ManyToMany annotation, with mapped by attribute. So now we have a bidirectional relationship where we can access any side of the relation.

```
      @ManyToMany( cascade= CascadeType.PERSIST ,mappedBy = "user")
      private List<User> users;
```

5. Todo

This part will be given to you by the teacher assistant in the lab time.

Experiments APKs

EXP. No.	Experiment Name	APK
1	Introduction to Android Programming	Link
2	Android Layouts	Link
3	Using Intents and Notifications	Link
4	SQLite Database	Link
5	Frame Animation and Tween Animation in Android	Frame-Link Tween-Link
6	Singleton and Shared Preferences	Link
7	Fragments	Fragments-Communication-Link Fragments-Transaction-Link
8	Integrating REST API into Android Application	Link

Note: Those APKs were tested using Pixel 3a XL device with API level 26