By Michael Franck

**Overview**

REST and CRUD (Create, Read, Update and Delete) operations are among some of the architect use for Web API/Data management. Students will develop a micro service using REST and CRUD to build a simple system that acts as a ledger for financial transactions.

**Learning Objectives**

After building the ledger project micro service, students should be able to:

* Setup a spring boot application
* Implement crud operation to store persistently to a MYSQL database
* Implement restful end points
* Use the command line tool curl to send Restful operation to a micro service

**Assumptions**

* Intro to programming and Java fundamentals
* Basics of object-oriented programming
* Data structures, collections, and exception handling
* Ability to utilize Spring Data JPA and Spring JdbcTemplates
* SQL, Joins, and normalization
* Understanding of RESTFUL APIs
* Spring MVC

**Resources and Preparation**

* MYSQL database server (8+)
* MYSQL Work bench (to run SQL’s to create database/table)
* CURL command line tool
* Java 1.8
* Gradle (5.6 or greater) build tool
* Java supported IDE (Example: Intellij/Net beans)
* DATABASE\_SCHEMA.sql (inside of project files)

**Instructor Walkthrough**

**Database Schema Setup (2 – 5 minutes)**

Open MYSQL work bench and connect to the MYSQL instance

Run the DATABASE\_SCHEMA.sql to create the project database/table.

**Code implementation (20 – 50 minutes)**

Open the java supported IDE and create a new project name: ledger\_project

Create the file “build.gradle” in add the following content (contains gradle configuration/tasks):

plugins {

id 'org.springframework.boot' version '2.4.1'

id 'io.spring.dependency-management' version '1.0.10.RELEASE'

id 'java'

id 'application'

}

mainClassName = 'com.ledger.app.Application'

group = 'com.ledger'

version = '0.0.1-SNAPSHOT'

sourceCompatibility = '1.8'

repositories {

mavenCentral()

}

dependencies {

implementation 'org.springframework.boot:spring-boot-starter-data-jpa'

implementation 'org.springframework.boot:spring-boot-starter-web'

implementation 'mysql:mysql-connector-java:8.0.21'

testImplementation 'org.springframework.boot:spring-boot-starter-test'

}

test {

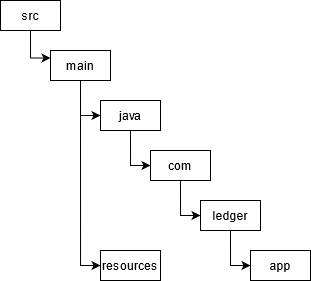
useJUnitPlatform()

}

Create the file “settings.gradle” in add the following content:

rootProject.name = 'app'

Create the following directory hierarchy:



Inside the resources directory, create a file with the name “application.properties” in add the below lines of code into it based of the database instance configuration:

database.driver=com.mysql.jdbc.Driver

database.url=

database.user=

database.password=

Example configuration:

database.driver=com.mysql.jdbc.Driver

database.url=jdbc:mysql://localhost:3306/transaction

database.user=root

database.password=password

Now, create a new java file with the name “Application” inside the app directory with the following content:

package com.ledger.app;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

At this point, we would have a spring application (which really doesn’t do anything at the moment). Before we continue, let’s run a gradle build command to make sure the project builds. From the command line (in the project root directory) run the following command:

gradlew build

If the gradle build ran successfully, you should get a build successful response. Below is an example:  


Next were going to create the file that reads the database configuration from the application.properties that we created earlier. Create the java file “AppConfig” inside of the app directory. Add the following content into it:

package com.ledger.app;

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

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.PropertySource;

import org.springframework.core.env.Environment;

import org.springframework.jdbc.datasource.DriverManagerDataSource;

import javax.sql.DataSource;

import java.util.Objects;

@Configuration

@PropertySource("classpath:application.properties")

public class AppConfig {

@Autowired

private Environment environment;

@Bean

DataSource dataSource() { // Get database configuration

DriverManagerDataSource driverManagerDataSource = new DriverManagerDataSource();

driverManagerDataSource.setUrl(environment.getProperty("database.url"));

driverManagerDataSource.setUsername(environment.getProperty("database.user"));

driverManagerDataSource.setPassword(environment.getProperty("database.password"));

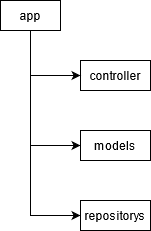
driverManagerDataSource.setDriverClassName(Objects.requireNonNull(environment.getProperty("database.driver")));

return driverManagerDataSource;

}

}

Create the following three sub directors in the app folder:



Before we get to the REST/CRUD implementation, we will need to create all the necessary classes (6 files) that are used by both implementation. So, create the java class “Transaction” inside of the models directory with the following content:

package com.ledger.app.models;

import java.math.BigDecimal;

public class Transaction {

public final Long id;

public final String sender;

public final String recipient;

public final BigDecimal transactionValue;

public Transaction(Long id, String sender, String recipient, BigDecimal transactionValue){

this.id = id;

this.sender = sender;

this.recipient = recipient;

this.transactionValue = transactionValue;

}

}

Create the java class “TransactionSum” inside of the models directory with the following content:

package com.ledger.app.models;

import java.math.BigDecimal;

public class TransactionSum {

public final BigDecimal sum;

public TransactionSum(BigDecimal sum){

this.sum = sum;

}

}

Create the java class “TransactionToAdd” inside of the models directory with the following content:

package com.ledger.app.models;

import java.math.BigDecimal;

public class TransactionToAdd {

public final String sender;

public final String recipient;

public final BigDecimal transactionValue;

TransactionToAdd(String sender, String recipient, BigDecimal transactionValue){

this.sender = sender;

this.recipient = recipient;

this.transactionValue = transactionValue;

}

}

Create the java class “TransactionUpdatedValue” inside of the models directory with the following content:

package com.ledger.app.models;

import java.math.BigDecimal;

public class TransactionUpdatedValue {

public BigDecimal transactionValue;

public TransactionUpdatedValue(){ }

public TransactionUpdatedValue(BigDecimal transactionValue){

this.transactionValue = transactionValue;

}

}

Create the java class “TransactionStatus” inside of the models directory with the following content:

package com.ledger.app.models;

public class TransactionStatus {

public final boolean success;

public final String message;

public TransactionStatus(boolean success, String message){

this.success = success;

this.message = message;

}

}

Create the java class “TransactionNotFoundException” inside of the models directory with the following content:

package com.ledger.app.models;

public class TransactionNotFoundException extends RuntimeException {

public TransactionNotFoundException(Long id){

super("Could not find transaction with id: " + id);

}

}

Now that we setup the necessary classes, we can now get to the CRUD/REST implementation. We will start with the CRUD implementation since the REST implementation depends on it. Create the java class “TransactionRepository” in the directory repositorys with the following content:

package com.ledger.app.repositorys;

import com.ledger.app.models.Transaction;

import com.ledger.app.models.TransactionToAdd;

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

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.stereotype.Repository;

import javax.sql.DataSource;

import java.math.BigDecimal;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.util.List;

@Repository

public class TransactionRepository {

private final JdbcTemplate jdbcTemplate;

// Database columns name

private final String TABLE = "transactions";

private final String COLUMN\_ID = "id";

private final String COLUMN\_SENDER = "sender";

private final String COLUMN\_RECIPIENT = "recipient";

private final String COLUMN\_TRANSACTION\_VALUE = "transaction\_value";

private final String COLUMN\_SOFT\_DELETE = "soft\_delete";

@Autowired

public TransactionRepository(DataSource dataSource) {

this.jdbcTemplate = new JdbcTemplate(dataSource);

}

/\*\*

\* Query the database for a single transaction row (object)

\* @param id The transaction id

\* @return A nullable transaction object

\*/

public Transaction findOne(long id) {

String sqlQuery = "select " + COLUMN\_ID +

", " + COLUMN\_SENDER +

", " + COLUMN\_RECIPIENT +

", " + COLUMN\_TRANSACTION\_VALUE +

" from " + TABLE + " where id = ?" + // The ? is an argument field which is replace with the id value in the below line

" and " + COLUMN\_SOFT\_DELETE + " = false";

return jdbcTemplate.query(sqlQuery, this::rowToEmployee, id) // Returns a list of transactions

.stream()

.findFirst() // Find and get the first transaction in the list

.orElse(null); // If the list is empty, return null

}

/\*\*

\* Query the database for all transactions

\* @return A list of transactions

\*/

public List<Transaction> findAll() {

String sqlQuery = "select " + COLUMN\_ID +

", " + COLUMN\_SENDER +

", " + COLUMN\_RECIPIENT +

", " + COLUMN\_TRANSACTION\_VALUE +

" from " + TABLE + " where " + COLUMN\_SOFT\_DELETE + " = false";

return jdbcTemplate.query(sqlQuery, this::rowToEmployee);

}

/\*\*

\* Inserts a new transaction into the database

\* @param newTransaction Transaction to add

\* @return An integer value representing if a record has been added

\*/

public int add(TransactionToAdd newTransaction){

String sqlQuery = "insert into " + TABLE + " (" + COLUMN\_SENDER +

", " + COLUMN\_RECIPIENT +

", " + COLUMN\_TRANSACTION\_VALUE +

") values (?, ?, ?)";

return jdbcTemplate.update(

sqlQuery,

newTransaction.sender,

newTransaction.recipient,

newTransaction.transactionValue

);

}

/\*\*

\* Update the transaction value

\* @param id The transaction id

\* @param value Transaction value

\* @return An integer value representing if a record has been updated

\*/

public int updateValue(Long id, BigDecimal value){

String sqlQuery = "update " + TABLE + " set " +

COLUMN\_TRANSACTION\_VALUE + " = ? where id = ?" +

" and " + COLUMN\_SOFT\_DELETE + " = false";

return jdbcTemplate.update(sqlQuery, value, id);

}

/\*\*

\* Updates the record soft delete column to true

\* @param id The transaction id

\* @return An integer value representing if a record has been updated

\*/

public int softDelete(Long id){

String sqlQuery = "update " + TABLE + " set " +

COLUMN\_SOFT\_DELETE + " = true where id = ?";

return jdbcTemplate.update(sqlQuery, id);

}

/\*\*

\* Transforms a database row to a Transaction object

\* @param resultSet Table data

\* @param rowNum

\* @return A nullable transaction object

\* @throws SQLException

\*/

private Transaction rowToEmployee(ResultSet resultSet, int rowNum) throws SQLException {

return new Transaction(

resultSet.getLong(COLUMN\_ID),

resultSet.getString(COLUMN\_SENDER),

resultSet.getString(COLUMN\_RECIPIENT),

resultSet.getBigDecimal(COLUMN\_TRANSACTION\_VALUE)

);

}

}

Now we can create the REST implementation. Create the java class “Ledger” in the controllers directory with the following content:

package com.ledger.app.controllers;

import com.ledger.app.models.\*;

import com.ledger.app.repositorys.TransactionRepository;

import org.springframework.http.HttpStatus;

import org.springframework.web.bind.annotation.\*;

import java.math.BigDecimal;

import java.util.List;

@RestController

public class Ledger {

private TransactionRepository repository;

Ledger(TransactionRepository repository){

this.repository = repository;

}

@GetMapping("/transaction")

List<Transaction> findAll(){

return repository.findAll();

}

@GetMapping("/transaction/sum")

TransactionSum getSumOfAll(){

BigDecimal sum = repository.findAll()

.stream()

.map(transaction -> transaction.transactionValue) // Get all the values for transactionValue

.reduce(BigDecimal.ZERO, BigDecimal::add);// Sums all the transactionValue(s), default zero if list is empty

return new TransactionSum(sum);

}

@GetMapping("/transaction/{id}")

Transaction findOne(@PathVariable Long id){

Transaction result = repository.findOne(id);

if(result != null){

return result;

} else {

throw new TransactionNotFoundException(id);

}

}

@PostMapping("/transaction")

TransactionStatus add(@RequestBody TransactionToAdd newTransaction){

int result = repository.add(newTransaction);

return new TransactionStatus(result > 0, null);

}

@PutMapping("/transaction/{id}")

TransactionStatus updateValue(@PathVariable Long id, @RequestBody TransactionUpdatedValue updatedValued){

int result = repository.updateValue(id, updatedValued.transactionValue);

if(result > 0) {

return new TransactionStatus(true, null);

} else {

throw new TransactionNotFoundException(id);

}

}

@DeleteMapping("/transaction/{id}")

TransactionStatus softDelete(@PathVariable Long id){

int result = repository.softDelete(id);

if(result > 0) {

return new TransactionStatus(true, null);

} else {

throw new TransactionNotFoundException(id);

}

}

@ResponseBody

@ExceptionHandler(TransactionNotFoundException.class) // Calls the below method whenever this exception is raised in this controller

@ResponseStatus(HttpStatus.NOT\_FOUND) // Set the HTTP status code to 404 for thr response

TransactionStatus transactionNotFoundHandler(TransactionNotFoundException ex) {

return new TransactionStatus(false, ex.getMessage());

}

}

At this point, we have all necessary class created to start the ledger micro service. From the command line inside the root directory, run the following commands:

gradlew build

gradlew run

If successful, the spring application will start up with no error.

**Interacting with the micro service**

Note: This is assuming that the database table is empty (not truncated)

Now that the micro service is up and running, let’s go ahead and start interacting with it. Let’s first try to get a list of transactions. Run the following curl command from the terminal:

curl -s -H "Content-Type: application/json" http://localhost:8080/transaction

The expected response is an empty array of: []. The reason is since the table was created, there’s no transaction record in it. We can fix that by adding a transaction record. To do that, run the following curl command:

curl -s -X POST -H "Content-Type: application/json" -d '{"sender": "testsender", "recipient": "testrecipient", "transactionValue": 15.99}' http://localhost:8080/transaction

Expected response:

{"success":true,"message":null}

Now, let’s try to get a list of transactions again:

curl -s -H "Content-Type: application/json" http://localhost:8080/transaction

Expected response:

[{"id":1,"sender":"testsender","recipient":"testrecipient","transactionValue":15.99}]

The response now returns a transaction recorder that we added earlier. We can also get this record by id (1) using the below command:

curl -s -H "Content-Type: application/json" http://localhost:8080/transaction/1

Expected response:

{"id":1,"sender":"testsender","recipient":"testrecipient","transactionValue":15.99}

Let’s add another transaction to test the sum operation. Run the following command:

curl -s -X POST -H "Content-Type: application/json" -d '{"sender": "testsender2", "recipient": "testrecipient2", "transactionValue": -5.00}' http://localhost:8080/transaction

Expected response:

{"success":true,"message":null}

What do you expect the output to be when make a request to get the sum of all the transaction?

Run the following command to get the sum of all the transactions:

curl -s -H "Content-Type: application/json" http://localhost:8080/transaction/sum

Expected response:

{"sum":10.99}

Let’s update one the first transaction value by id:

curl -s -X PUT -H "Content-Type: application/json" -d '{"transactionValue": 18.99}' http://localhost:8080/transaction/1

Expected response:

{"success":true,"message":null}

Let’s verify that the value was updated with the following command:

curl -s -H "Content-Type: application/json" http://localhost:8080/transaction/1

Expected response:

{"id":1,"sender":"testsender","recipient":"testrecipient","transactionValue":18.99}

Let’s delete (soft) the first transaction

curl -s -X DELETE -H "Content-Type: application/json" http://localhost:8080/transaction/1

Expected response:

{"success":true,"message":null}

Now if we pull a list of all the transaction, we should expect to see only one transaction:

curl -s -H "Content-Type: application/json" http://localhost:8080/transaction

Expected response:

[{"id":2,"sender":"testsender2","recipient":"testrecipient2","transactionValue":-5.0}]