Problem Statement

Create Employee CRUD operation and share API endpoint for Registration:

Employee can have

User Id (This should be unique for every Employee)

First Name

Last Name

Mobile Number

Address

Department

Write Http based rest API's using java spring boot to complete the below mentioned operations,

POST / Registration (create)

Request parameters are Firstname, Lastname, Mobile Number, Address, Department

Return the saved Employee Object as a JSON format

PATCH / Update

Path Variable: user_id

Need to update firstName

Return the updated Employee Object as a JSON format.

DELETE one record

Request parameter: user_id

Return Success Message ("User Deleted Successfully").

GET / list (List all employees)

Response will be List of Employees from the backend as a JSON format

1. Create MySQL Database

We will be developing RESTful APIs that allow clients to perform CRUD operations about products, so create the following table in MySQL server:
This is a simple table containing just 3 columns: id, name and price.

```
CREATE TABLE `product` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(45) NOT NULL,
  `price` float NOT NULL,
  PRIMARY KEY (`id`)
);
```

2. Configure Maven Dependencies

Specify the following configuration in your project's `pom.xml` file:

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.2.2.RELEASE</version>
</parent>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

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



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

We see, we use Spring Boot version 2.2.2.

The artifact `spring-boot-starter-web` is for Spring Web MVC, RESTful webservices and embedded Tomcat server.

The artifact `spring-boot-starter-data-jpa` is for Spring Data JPA and Hibernate.

The artifact `mysql-connector-java` is for JDBC driver for MySQL.

We don’t need to specify versions of these dependencies because Spring Boot uses the best and most compatible versions (thanks!).

TIP: Use Spring Boot DevTools for automatic restart so you don't have to manually restart the application during development.

3. Configure Data Source Properties

Next, we need to specify database connection information. So create the `application.properties` file under `src/main/resources` directory with the following content:

```
spring.jpa.hibernate.ddl-auto=none
spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=root
spring.datasource.password=password
```

Remember to update url, username and password according to your MySQL database server.

4. Code Domain Model Class

Next, create the `Product` class to map with the product table in the database as follows:

```
package net.codejava;

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

@Entity
public class Product {
    private Integer id;
    private String name;
    private float price;

    public Product() {
    }

    public Product(Integer id, String name, float price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public Integer getId() {
        return id;
    }

    // other getters and setters...
}
```

This is a simple domain model class, with class name and field names are identical to table name and column names in the database – so we can use the minimum number of JPA annotations.

5. Code Repository Interface

To take advantages of Spring Data JPA, create the `ProductRepository` interface as below:

```
package net.codejava;

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

public interface ProductRepository extends JpaRepository<Product, Integer> {

}
```

Then Spring Data JPA will generate implementation code for the most common CRUD operations – we don't have to write a single query.

6. Code Service Class

Next, code a class that acts as a middle layer between persistence layer (repository) and controller layer. Create the `ProductService` class with the following code:

```
package net.codejava;

import java.util.List;

import javax.transaction.Transactional;

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

@Service
@Transactional
public class ProductService {

    @Autowired
    private ProductRepository repo;

    public List<Product> listAll() {
        return repo.findAll();
    }

    public void save(Product product) {
        repo.save(product);
    }

    public Product get(Integer id) {
        return repo.findById(id).get();
    }

    public void delete(Integer id) {
        repo.deleteById(id);
    }
}
```

As you can see, this business/service class simply forwards the calls to an implementation of the `ProductRepository` interface. In addition, all the methods are executed in transactions because this class is marked with the `@Transactional` annotation.

7. Code Spring Boot Application Class

To run our Spring Boot application, we need to create the main class as shown below:

```
package net.codejava;

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);
    }

}
```

This class will start embedded Tomcat server hosting our Spring Boot web application.

8. Code REST Controller Class

Here, we come to the part that actually exposes RESTful APIs for CRUD operations – a Spring controller following REST style. Create the `ProductController` class with some initial code as below:

```java
package net.codejava;

import java.util.*;

import org.springframework.beans.factory.annotation.*;
import org.springframework.http.*;

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

@RestController
public class ProductController {

    @Autowired
    private ProductService service;

    // RESTful API methods for Retrieval operations

    // RESTful API method for Create operation

    // RESTful API method for Update operation

    // RESTful API method for Delete operation
}
```

The `@RestController` annotation enables the handler methods in this controller to be exposed as RESTful APIs, e.g. reading JSON data from the request and including JSON data to the response. And this REST controller still takes advantages of Spring's dependency injection (an instance of `ProductService` is autowired).

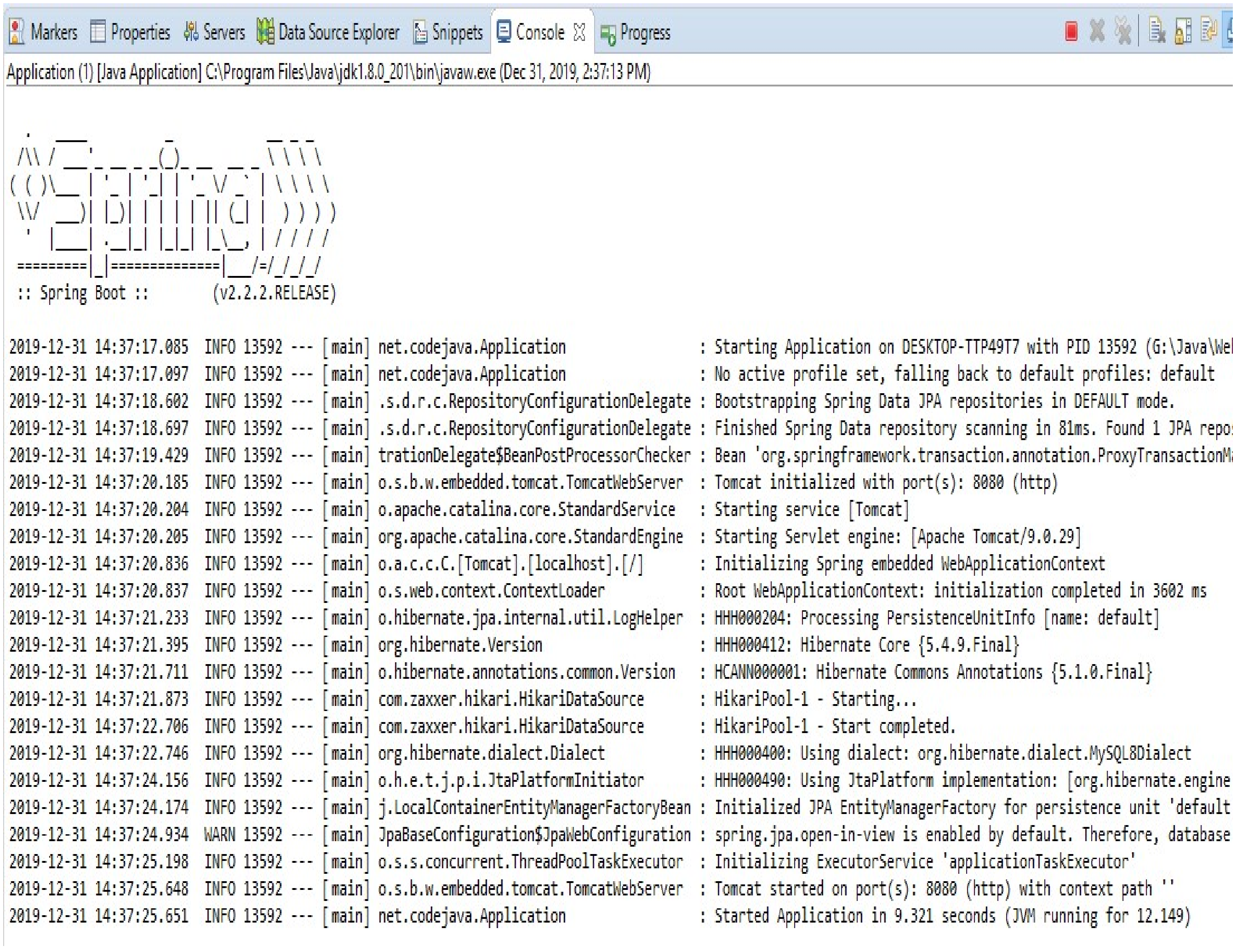
Let's see how to implement RESTful API for each CRUD operation in the next sections.

9. Implement RESTful API for Retrieval Operation

The first RESTful API is a method that returns a list of product (a kind of retrieval operation):

```
@GetMapping("/products")
public List<Product> list() {
    return service.listAll();
}
```

Now we can run the `Application` class and test. With Eclipse, you can see the Spring Boot logo appears in the Console view as shown in the following screenshot:



Application (1) [Java Application] C:\Program Files\Java\jdk1.8.0_201\bin\javaw.exe (Dec 31, 2019, 2:37:13 PM)

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.2.2.RELEASE)

2019-12-31 14:37:17.085  INFO 13592 --- [main] net.codejava.Application                 : Starting Application on DESKTOP-TTP49T7 with PID 13592 (G:\Java\We
2019-12-31 14:37:17.097  INFO 13592 --- [main] net.codejava.Application                 : No active profile set, falling back to default profiles: default
2019-12-31 14:37:18.602  INFO 13592 --- [main] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JPA repositories in DEFAULT mode.
2019-12-31 14:37:18.697  INFO 13592 --- [main] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 81ms. Found 1 JPA repo
2019-12-31 14:37:19.429  INFO 13592 --- [main] trationDelegate$BeanPostProcessorChecker : Bean 'org.springframework.transaction.annotation.ProxyTransactionM
2019-12-31 14:37:20.185  INFO 13592 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
2019-12-31 14:37:20.204  INFO 13592 --- [main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2019-12-31 14:37:20.205  INFO 13592 --- [main] org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9.0.29]
2019-12-31 14:37:20.836  INFO 13592 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2019-12-31 14:37:20.837  INFO 13592 --- [main] o.s.web.context.ContextLoader            : Root WebApplicationContext: initialization completed in 3602 ms
2019-12-31 14:37:21.233  INFO 13592 --- [main] o.hibernate.jpa.internal.util.LogHelper  : HHH000204: Processing PersistenceUnitInfo [name: default]
2019-12-31 14:37:21.395  INFO 13592 --- [main] org.hibernate.Version                    : HHH000412: Hibernate Core {5.4.9.Final}
2019-12-31 14:37:21.711  INFO 13592 --- [main] o.hibernate.annotations.common.Version   : HCANN000001: Hibernate Commons Annotations {5.1.0.Final}
2019-12-31 14:37:21.873  INFO 13592 --- [main] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Starting...
2019-12-31 14:37:22.706  INFO 13592 --- [main] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Start completed.
2019-12-31 14:37:22.746  INFO 13592 --- [main] org.hibernate.dialect.Dialect            : HHH000400: Using dialect: org.hibernate.dialect.MySQL8Dialect
2019-12-31 14:37:24.156  INFO 13592 --- [main] o.h.e.t.j.p.i.JtaPlatformInitiator       : HHH000490: Using JtaPlatform implementation: [org.hibernate.engine
2019-12-31 14:37:24.174  INFO 13592 --- [main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default
2019-12-31 14:37:24.934  WARN 13592 --- [main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database
2019-12-31 14:37:25.198  INFO 13592 --- [main] o.s.s.concurrent.ThreadPoolTaskExecutor  : Initializing ExecutorService 'applicationTaskExecutor'
2019-12-31 14:37:25.648  INFO 13592 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
2019-12-31 14:37:25.651  INFO 13592 --- [main] net.codejava.Application                 : Started Application in 9.321 seconds (JVM running for 12.149)
```

From the logging information, you can see Tomcat, Spring Data JPA and Hibernate are started. And the server listens on port 8080 by default.

Use MySQL Workbench to insert the first row into the product table (name: iPhone X, price: 999.98). And type the following curl command:

```
curl http://localhost:8080/products
```

Then you can see the server's response like this:

```
[{"id":1,"name":"iPhone X","price":999.98}]
```

This is the JSON representation of a List of Product objects. Spring uses Jackson JSON library to serialize Java objects to JSON and vice-versa, automatically.

Next, let's implement the second method for a RESTful API that allows the clients to get information about a specific product based on ID, like this:

```
@GetMapping("/products/{id}")
public ResponseEntity<Product> get(@PathVariable Integer id) {
    try {
        Product product = service.get(id);
        return new ResponseEntity<Product>(product, HttpStatus.OK);
    } catch (NoSuchElementException e) {
        return new ResponseEntity<Product>(HttpStatus.NOT_FOUND);
    }
}
```

In this method, if a product is found for the given ID, the server sends a response that includes JSON representation of the Product object with HTTP status OK (200). Else if no product is found, it returns HTTP status Not Found (404).

Type the following curl command to test:

```
curl http://localhost:8080/products/1
```

Response from the server:

```
[{"id":1,"name":"iPhone X","price":999.98}]
```

Try to test with a non-exist ID:

```
curl -v http://localhost:8080/products/10
```

You will see HTTP 404 status in the verbose output.

NOTE: In case of no items in the list (the returned list is empty) the API should return HTTP

```
curl -X POST -H "Content-Type: application/json" -d "{\"name\":\"XBox
360\",\"price\":299.99}" http://localhost:8080/products
```

status 204 No Content. For details, check this article: Spring Boot REST API CRUD with HATEOAS Tutorial

10. Implement RESTful API for Create Operation

The following method will expose a RESTful API that allows the clients to create a product

```
@PostMapping("/products")
public void add(@RequestBody Product product) {
    service.save(product);
}
```

Using `@PostMapping` annotation, this method handles only HTTP POST request. And type the following command to test.

Check the database in MySQL Workbench and you should see a second row was inserted.

For validating data in the request's body, refer to this article: Spring Boot REST API Request Validation Examples

NOTE: if the creation operation is successful, the API should return HTTP status 201 Created. For details, check this article: Spring Boot REST API CRUD with HATEOAS Tutorial

11. Implement RESTful API for Update Operation

Write code for the method that exposes RESTful API for update operation as follows:

```
@PutMapping("/products/{id}")
public ResponseEntity<?> update(@RequestBody Product product, @PathVariable Integer id) {
    try {
        Product existProduct = service.get(id);
        service.save(product);
        return new ResponseEntity<>(HttpStatus.OK);
    } catch (NoSuchElementException e) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
}
```

The `@PutMapping` annotation specifies that this method accepts only HTTP PUT request. If a product found with the given ID, it is updated and the server returns HTTP status OK. If no product found, the HTTP status Not Found (404) is returned.
Using curl, type the following command to test updating the product with ID 1:
Type this command to check the update:

```
curl http://localhost:8080/products
```

12. Implement RESTful API for Delete Operation

Finally, we implement the method that exposes RESTful API for the delete operation as simple as follows:

```
@DeleteMapping("/products/{id}")
public void delete(@PathVariable Integer id) {
    service.delete(id);
}
```

And type the following command to test deleting the product with ID 1:

```
1curl -X DELETE http://localhost:8080/products/1
```

You should see the row ID 1 removed from the database.
NOTE: if the delete operation is successful, the API should return HTTP status code 204 No Content. For details, check this article: Spring Boot REST API CRUD with HATEOAS

```
curl -X PUT -H "Content-Type: application/json" -d
"{\"id\":1,\"name\":\"iPad\",\"price\":888}" http://localhost:8080/products/1
```

Tutorial.
Congratulations, you have completed the Spring Boot tutorial about RESTful APIs for CRUD operations with MySQL database. For your reference, you can download the sample project attached below.