Spring Boot REST API CRUD Example With MySQL Database

mysql spring boot spring boot 3 spring data jpa tutorial

This tutorial will teach you how to build CRUD REST APIs using Spring Boot 3, Spring Data JPA, and MySQL Database.

We'll first build the APIs to create, retrieve, update and delete a user, then test them using postman.

Note that we are using the latest version of Spring boot which is version 3.
Learn Spring boot at **https://www.javaguides.net/p/spring-boot-tutorial.html**.

1. Create a Spring Boot Application and Import in IntelliJ IDEA

You can use the Spring Initializer website (**start.spring.io**) or the Spring Boot CLI to generate a new Spring Boot project with the necessary dependencies.

Refer to the below screenshot to enter details while creating the spring boot application using the **spring initializr**:

Click on Generate button to download the Spring boot project as a zip file. Unzip the zip file and import the Spring boot project in IntelliJ IDEA.

Here is the pom.xml file for your reference:

```
<?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
https://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>3.0.0-M4</version>
                <relativePath/> <!-- lookup parent from repository -->
        </parent>
        <groupId>net.javaguides</groupId>
        <artifactId>springboot-restful-webservices</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <name>springboot-restful-webservices</name>
        <description>Demo project for Spring Boot Restful
Webservices</description>
        <properties>
                <java.version>17</java.version>
        </properties>
        <dependencies>
                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-data-jpa</artifactId>
                </dependency>
                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-web</artifactId>
                </dependency>

                <dependency>
                        <groupId>mysql</groupId>
                        <artifactId>mysql-connector-java</artifactId>
                        <scope>runtime</scope>
                </dependency>
                <dependency>
                        <groupId>org.projectlombok</groupId>
                        <artifactId>lombok</artifactId>
                        <optional>true</optional>
                </dependency>
                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-test</artifactId>
                        <scope>test</scope>
                </dependency>
        </dependencies>

        <build>
                <plugins>
                        <plugin>
                                <groupId>org.springframework.boot</groupId>
                                <artifactId>spring-boot-maven-
plugin</artifactId>
                                <configuration>
                                        <excludes>
                                                <exclude>

        <groupId>org.projectlombok</groupId>

        <artifactId>lombok</artifactId>
                                                </exclude>
                                        </excludes>
                                </configuration>
                        </plugin>
                </plugins>
```

```
        </build>
        <repositories>
                <repository>
                        <id>spring-milestones</id>
                        <name>Spring Milestones</name>
                        <url>https://repo.spring.io/milestone</url>
                        <snapshots>
                                <enabled>false</enabled>
                        </snapshots>
                </repository>
        </repositories>
        <pluginRepositories>
                <pluginRepository>
                        <id>spring-milestones</id>
                        <name>Spring Milestones</name>
                        <url>https://repo.spring.io/milestone</url>
                        <snapshots>
                                <enabled>false</enabled>
                        </snapshots>
                </pluginRepository>
        </pluginRepositories>

</project>
```

2. Project Structure

Refer to the below screenshot to create a project structure or a packing structure for our Spring boot application:

3. Configuring MySQL Database

Since we're using MySQL as our database, we need to configure the URL, username, and password so that our Spring boot can establish a connection with the database on startup. Open the `src/main/resources/application.properties` file and add the following properties to it:

```
spring.datasource.url=jdbc:mysql://localhost:3306/user_management
spring.datasource.username=root
spring.datasource.password=Mysql@123

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
spring.jpa.hibernate.ddl-auto=update
```

Don't forget to change

the `spring.datasource.username` and `spring.datasource.password` as per your MySQL installation. Also, create a database named **user_management** in MySQL before proceeding to the next section.

You don't need to create any tables. The tables will automatically be created by Hibernate from the `User` entity that we will define in the next step. This is made possible by the property `spring.jpa.hibernate.ddl-auto = update.`

4. Create JPA Entity - User.java

An Entity is a plain old Java object (POJO) that represents the data you want to store. You will need to annotate the class with *`@Entity`* and define the fields of the class along with the getters and setters for each field.

Let's create a `User` JPA entity class with the following fields:

- id - primary key
- firstName - user first name
- lastName - user last name
- email - user email ID

```
package net.javaguides.springboot.entity;

import jakarta.persistence.*;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Entity
@Table(name = "users")
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    private Long id;
    @Column(nullable = false)
    private String firstName;
    @Column(nullable = false)
    private String lastName;
    @Column(nullable = false, unique = true)
    private String email;
}
```

Note that we are using Lombok annotations to reduce the boilerplate code such as getter/setter methods, and constructors.

We are using below JPA annotations to map an Entity with a database table:

`@Entity` annotation is used to mark the class as a persistent Java class.

`@Table` annotation is used to provide the details of the table that this entity will be mapped to.

`@Id` annotation is used to define the primary key.

`@GeneratedValue` annotation is used to define the primary key generation strategy. In the above case, we have declared the primary key to be an Auto Increment field.

`@Column` annotation is used to define the properties of the column that will be mapped to the annotated field. You can define several properties like name, length, nullable, updateable, etc.

5. Create Spring Data JPA Repository for User JPA Entity

Let's create a `UserRepository` to access the User's data from the database.

Well, Spring Data JPA comes with a `JpaRepository` interface that defines methods for all the CRUD operations on the entity, and a default implementation of `JpaRepository` called `SimpleJpaRepository`.

```
package net.javaguides.springboot.repository;
```

```
import net.javaguides.springboot.entity.User;
import org.springframework.data.jpa.repository.JpaRepository;

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

6. Service Layer Implementation

This layer will contain the business logic for the API and will be used to perform CRUD operations using the Repository.

UserService Interface

```
package net.javaguides.springboot.service;

import net.javaguides.springboot.entity.User;

import java.util.List;

public interface UserService {
    User createUser(User user);

    User getUserById(Long userId);

    List<User> getAllUsers();

    User updateUser(User user);

    void deleteUser(Long userId);
}
```

UserServiceImpl Class

```
package net.javaguides.springboot.service.impl;

import lombok.AllArgsConstructor;
import net.javaguides.springboot.entity.User;
import net.javaguides.springboot.repository.UserRepository;
import net.javaguides.springboot.service.UserService;
import org.apache.logging.log4j.util.Strings;
import org.springframework.stereotype.Service;
import org.springframework.util.StringUtils;

import java.util.List;
import java.util.Objects;
import java.util.Optional;
```

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

    private UserRepository userRepository;

    @Override
    public User createUser(User user) {
        return userRepository.save(user);
    }

    @Override
    public User getUserById(Long userId) {
        Optional<User> optionalUser = userRepository.findById(userId);
        return optionalUser.get();
    }

    @Override
    public List<User> getAllUsers() {
        return userRepository.findAll();
    }

    @Override
    public User updateUser(User user) {
        User existingUser = userRepository.findById(user.getId()).get();
        existingUser.setFirstName(user.getFirstName());
        existingUser.setLastName(user.getLastName());
        existingUser.setEmail(user.getEmail());
        User updatedUser = userRepository.save(existingUser);
        return updatedUser;
    }

    @Override
    public void deleteUser(Long userId) {
        userRepository.deleteById(userId);
    }
}
```

7. Creating UserController - Building CRUD Rest APIs

The controller is responsible for handling incoming HTTP requests and returning the appropriate response. You will need to define the endpoints of the API and map them to the appropriate methods in the Service layer.

Let's create the REST APIs for creating, retrieving, updating, and deleting a `User`:

```java
package net.javaguides.springboot.controller;

import lombok.AllArgsConstructor;
import net.javaguides.springboot.entity.User;
import net.javaguides.springboot.service.UserService;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@AllArgsConstructor
@RequestMapping("api/users")
public class UserController {

    private UserService userService;

    // build create User REST API
    @PostMapping
    public ResponseEntity<User> createUser(@RequestBody User user){
        User savedUser = userService.createUser(user);
        return new ResponseEntity<>(savedUser, HttpStatus.CREATED);
    }

    // build get user by id REST API
    // http://localhost:8080/api/users/1
    @GetMapping("{id}")
    public ResponseEntity<User> getUserById(@PathVariable("id") Long userId){
        User user = userService.getUserById(userId);
        return new ResponseEntity<>(user, HttpStatus.OK);
    }

    // Build Get All Users REST API
    // http://localhost:8080/api/users
    @GetMapping
    public ResponseEntity<List<User>> getAllUsers(){
        List<User> users = userService.getAllUsers();
        return new ResponseEntity<>(users, HttpStatus.OK);
    }

    // Build Update User REST API
    @PutMapping("{id}")
    // http://localhost:8080/api/users/1
    public ResponseEntity<User> updateUser(@PathVariable("id") Long userId,
                                           @RequestBody User user){
        user.setId(userId);
        User updatedUser = userService.updateUser(user);
        return new ResponseEntity<>(updatedUser, HttpStatus.OK);
    }

    // Build Delete User REST API
    @DeleteMapping("{id}")
    public ResponseEntity<String> deleteUser(@PathVariable("id") Long userId){
        userService.deleteUser(userId);
        return new ResponseEntity<>("User successfully deleted!", HttpStatus.OK);
    }
```

```
}
```

8. Running the Application

We have successfully developed all the CRUD Rest APIs for the `User` model. Now it's time to deploy our application in a servlet container(embedded tomcat).

Two ways we can start the standalone Spring boot application.

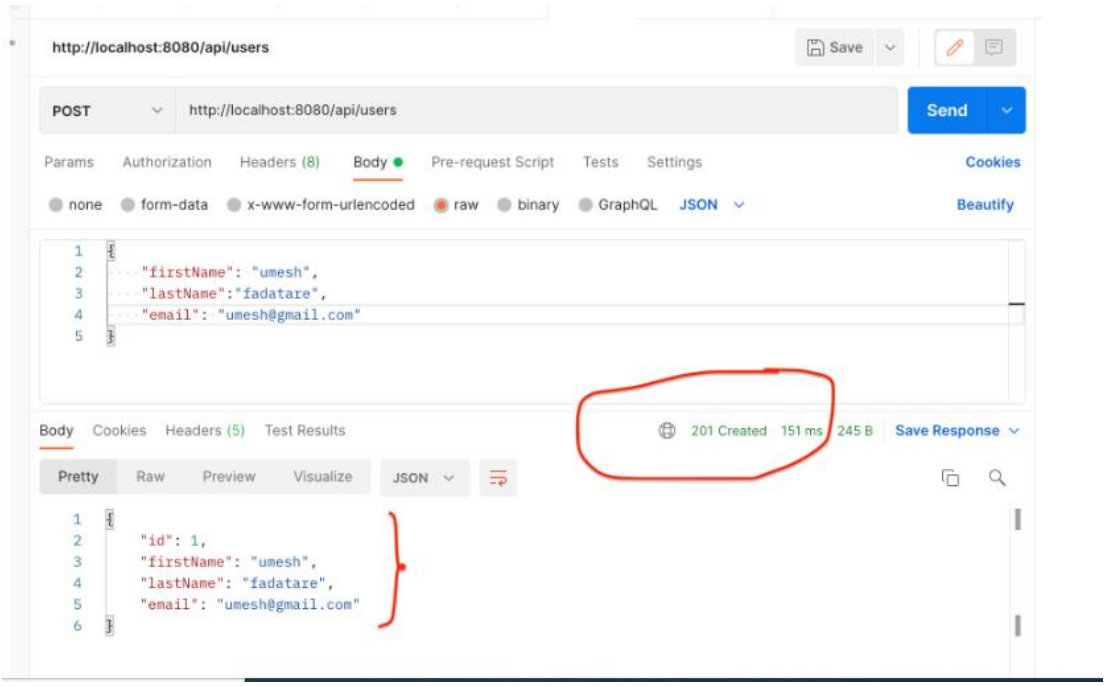
1. From the root directory of the application and type the following command to run it -

```
$ mvn spring-boot:run
```

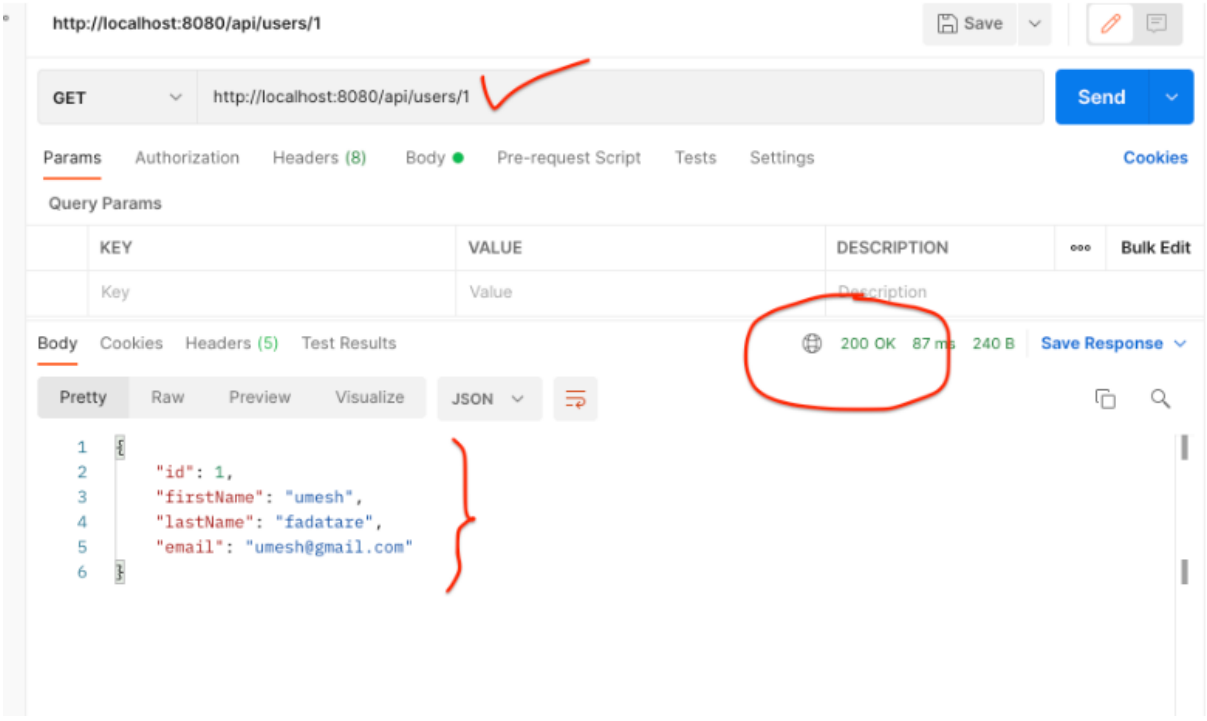
2. From your IDE, run the SpringbootRestfulWebservicesApplication.main() method as a standalone Java class that will start the embedded Tomcat server on port 8080 and point the browser to **http://localhost:8080/**.

9. Test Spring Boot CRUD REST APIs using Postman Client

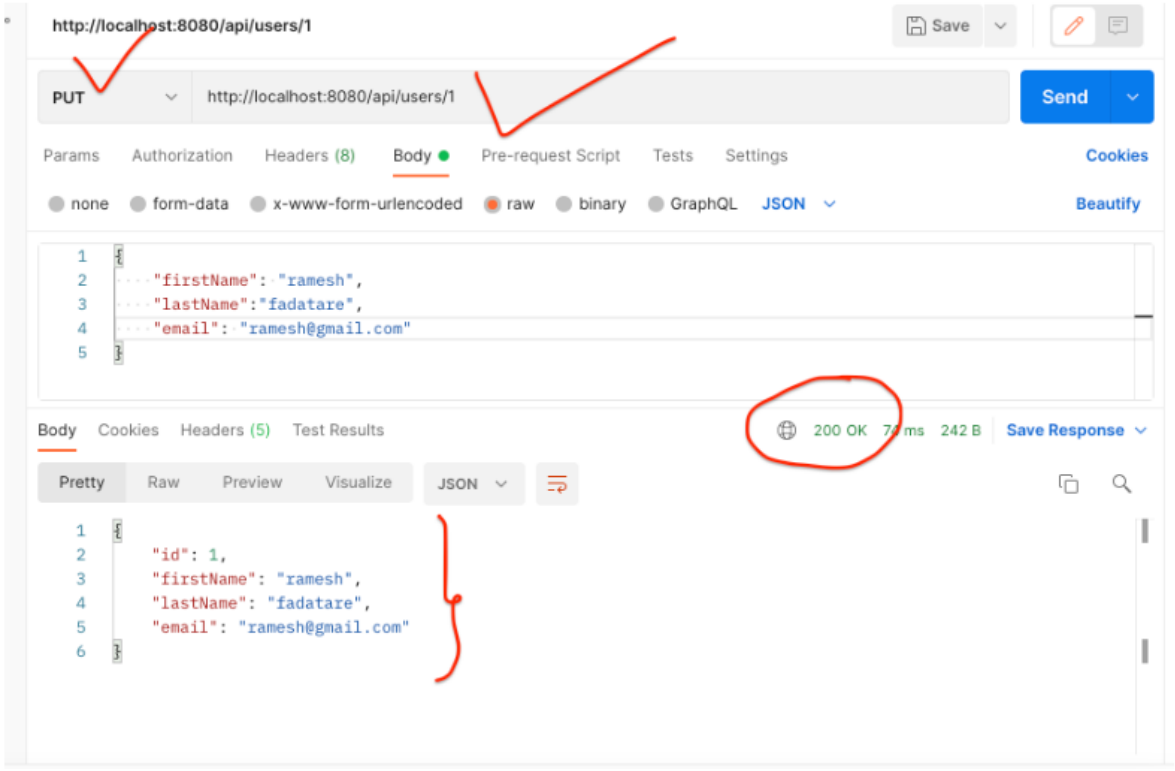
Create User REST API:



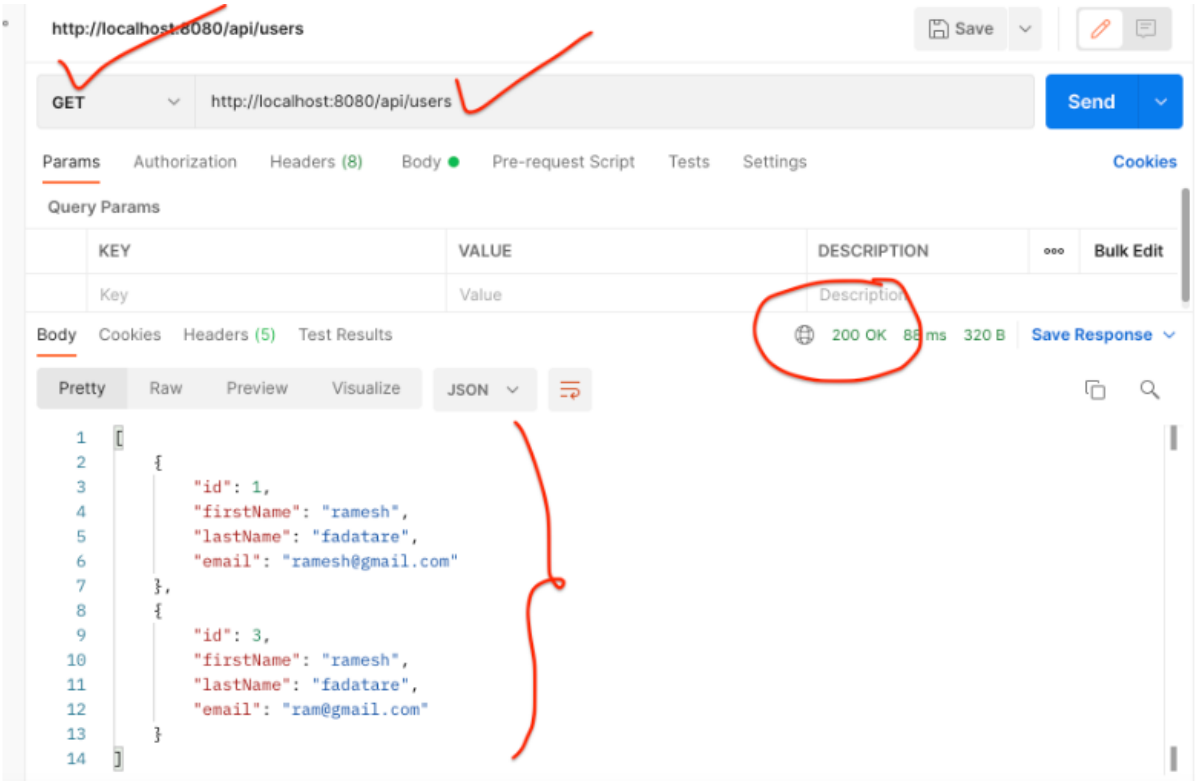
Get Single User REST API:



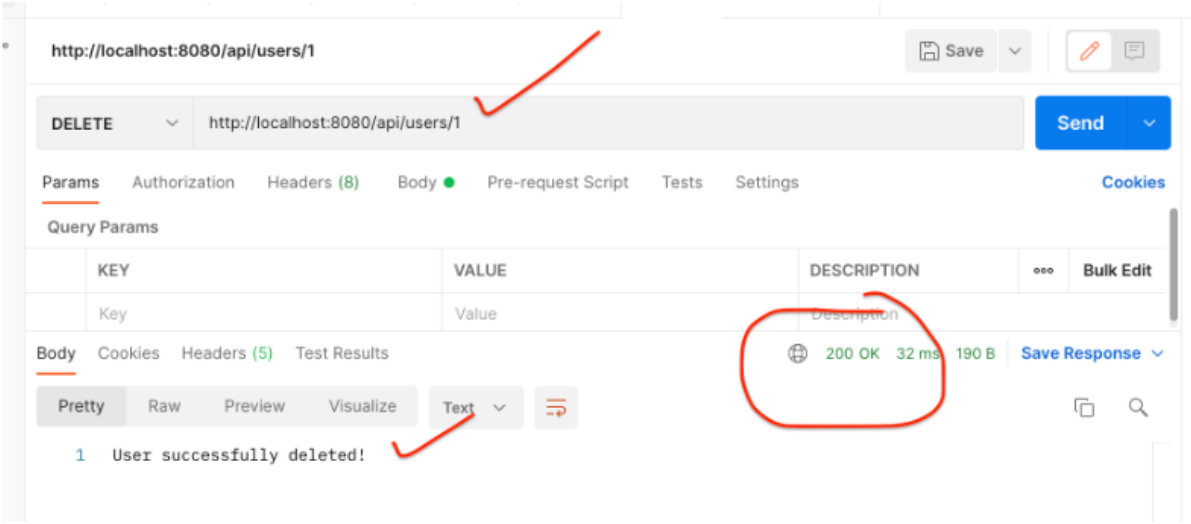
Update User REST API:



Get All Users REST API:



Delete User REST API:



10. Conclusion

Congratulations guys! We successfully built a Restful CRUD API using Spring Boot 3, Spring Data JPA, and MySQL database.