High-level architecture of Spring boot project



1. Creating a Spring Boot Application

There are many ways to create a Spring Boot application. You can refer below articles to create a Spring Boot application.

* >> Create Spring Boot Project With Spring Initializer
* >> Create Spring Boot Project in Spring Tool Suite [STS]

Refer to the next step to create a project packaging structure.

2. Maven dependencies

Let's define all required maven dependencies in pom.xml file:

<?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>2.2.6.RELEASE</version>

<relativePath/>

<!-- lookup parent from repository -->

</parent>

<groupId>net.javaguides</groupId>

<artifactId>springboot-crud-restful-webservices</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>springboot-crud-restful-webservices</name>

<description>Demo project for Spring Boot Restful web services</description>

<properties>

<java.version>1.8</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.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

<exclusions>

<exclusion>

<groupId>org.junit.vintage</groupId>

<artifactId>junit-vintage-engine</artifactId>

</exclusion>

</exclusions>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

3. Configuring MySQL Database

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

spring.datasource.url = jdbc:mysql://localhost:3306/usersDB?useSSL=false

spring.datasource.username = root

spring.datasource.password = root

## Hibernate Properties

# The SQL dialect makes Hibernate generate better SQL for the chosen database

spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5InnoDBDialect

# Hibernate ddl auto (create, create-drop, validate, update)

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 usersDB 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 User JPA Entity

Let's create a User model or domain class with the following fields:

id - primary key

firstName - user first name

lastName - user last name

emailId - user email ID

.............

package net.javaguides.springboot.entity;

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 = "users")

public class User {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private long id;

@Column(name = "first\_name")

private String firstName;

@Column(name = "last\_name")

private String lastName;

@Column(name = "email")

private String email;

public User() {

}

public User(String firstName, String lastName, String email) {

super();

this.firstName = firstName;

this.lastName = lastName;

this.email = email;

}

public long getId() {

return id;

}

public void setId(long id) {

this.id = id;

}

public String getFirstName() {

return firstName;

}

public void setFirstName(String firstName) {

this.firstName = firstName;

}

public String getLastName() {

return lastName;

}

public void setLastName(String lastName) {

this.lastName = lastName;

}

public String getEmail() {

return email;

}

public void setEmail(String email) {

this.email = email;

}

}

All your domain models must be annotated with @Entity annotation. It 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. Define UserRepository

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

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

...................

package net.javaguides.springboot.repository;

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

import org.springframework.stereotype.Repository;

import net.javaguides.springboot.entity.User;

@Repository

public interface UserRepository extends JpaRepository<User, Long>{

}

6. Creating Custom Business Exception

We’ll define the Rest APIs for creating, retrieving, updating, and deleting a User in the next step.

The APIs will throw a ResourceNotFoundException whenever a User with a given id is not found in the database.

Following is the definition of ResourceNotFoundException:

..............................................

package net.javaguides.springboot.exception;

import org.springframework.http.HttpStatus;

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

@ResponseStatus(value = HttpStatus.NOT\_FOUND)

public class ResourceNotFoundException extends RuntimeException {

private static final long serialVersionUID = 1 L;

public ResourceNotFoundException(String message) {

super(message);

}

}

7. Creating UserController - Building CRUD Rest APIs

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

package net.javaguides.springboot.controller;

import java.util.List;

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

import org.springframework.http.ResponseEntity;

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

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

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

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

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

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

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

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

import net.javaguides.springboot.entity.User;

import net.javaguides.springboot.exception.ResourceNotFoundException;

import net.javaguides.springboot.repository.UserRepository;

@RestController

@RequestMapping("/api/users")

public class UserController {

@Autowired

private UserRepository userRepository;

// get all users

@GetMapping

public List < User > getAllUsers() {

return this.userRepository.findAll();

}

// get user by id

@GetMapping("/{id}")

public User getUserById(@PathVariable(value = "id") long userId) {

return this.userRepository.findById(userId)

.orElseThrow(() - > new ResourceNotFoundException("User not found with id :" + userId));

}

// create user

@PostMapping

public User createUser(@RequestBody User user) {

return this.userRepository.save(user);

}

// update user

@PutMapping("/{id}")

public User updateUser(@RequestBody User user, @PathVariable("id") long userId) {

User existingUser = this.userRepository.findById(userId)

.orElseThrow(() - > new ResourceNotFoundException("User not found with id :" + userId));

existingUser.setFirstName(user.getFirstName());

existingUser.setLastName(user.getLastName());

existingUser.setEmail(user.getEmail());

return this.userRepository.save(existingUser);

}

// delete user by id

@DeleteMapping("/{id}")

public ResponseEntity < User > deleteUser(@PathVariable("id") long userId) {

User existingUser = this.userRepository.findById(userId)

.orElseThrow(() - > new ResourceNotFoundException("User not found with id :" + userId));

this.userRepository.delete(existingUser);

return ResponseEntity.ok().build();

}

}

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 SpringBootCrudRestApplication.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/>.