**1. Spring Framework**

Spring is a very popular JavaEE framework for building web and enterprise applications.

If you are a Java developer then there is a high chance that you might have heard about the Spring framework and probably have used it in your projects.

Spring is very popular for several reasons:

* Spring’s dependency injection approach encourages writing testable code
* Easy to use but powerful database transaction management capabilities
* Spring simplifies integration with other Java frameworks like JPA/Hibernate ORM, Struts/JSF/etc. web frameworks
* State of the art Web MVC framework for building web applications

Basically, Spring-based applications have a lot of configurations. For example:

When we develop Spring MVC web application using Spring MVC then we need to configure

- Component scan,

- Dispatcher Servlet

- View resolver

- Web jars(for delivering static content) among other things.

When we use Hibernate/JPA in the same Spring MVC application then we would need to configure a

- Data source

- Entity manager factory/session factory

- Transaction manager among a host of other things.

When you use cache in the same Spring MVC application then we need to configure

- Cache configuration

When you use Message Queue in the same Spring MVC application then we need to configure

- Message queue configuration

When you use a NoSQL database in the same Spring MVC application then we need to configure

- NoSQL database configuration

A lot of configuration right.

If Spring can automatically do it for me that would be awesome!!!.

Well, Spring Boot does what exactly you are looking for. It will do things automatically for you but allows you to override the defaults if you want to.

The Spring team created Spring Boot to address the complexity of the configuration.

**2. What is Spring Boot?**

Spring Boot is basically an extension of the Spring framework which eliminated the boilerplate configurations required for setting up a Spring application.  
  
Spring Boot is an opinionated framework that helps developers build Spring-based applications quickly and easily. **The main goal of Spring Boot is to quickly create Spring-based applications without requiring developers to write the same boilerplate configuration again and again.**

**3. Key Spring Boot features**

Let me a list of a few key features of the Spring boot and we will discuss each key feature briefly.

1. Spring Boot starters
2. Spring Boot autoconfiguration
3. Elegant configuration management
4. Spring Boot actuator
5. Easy-to-use embedded servlet container support

**3.1. Spring Boot Starters**

Spring Boot offers many starter modules to get started quickly with many of the commonly used technologies, like SpringMVC, JPA, MongoDB, Spring Batch, SpringSecurity, Solr, ElasticSearch, etc. These starters are pre-configured with the most commonly used library dependencies so you don’t have to search for the compatible library versions and configure them manually.

For example, the spring-boot-starter-data-jpa starter module includes all the dependencies required to use Spring Data JPA, along with Hibernate library dependencies, as Hibernate is the most commonly used JPA implementation.

One more example, when we add the spring-boot-starter-web dependency, it will by default pull all the commonly used libraries while developing Spring MVC applications, such as spring-webmvc, jackson-json, validation-api, and tomcat.

Not only does the spring-boot-starter-web add all these libraries but it also configures the commonly registered beans like DispatcherServlet, ResourceHandlers, MessageSource, etc. with sensible defaults.

Read more about starters on [**Important Spring boot Starters with Examples**](http://www.javaguides.net/2018/09/important-spring-boot-starters-with-examples.html)

**3.2. Spring Boot Autoconfiguration**

Spring Boot addresses the problem that Spring applications need complex configuration by eliminating the need to manually set up the boilerplate configuration.

Spring Boot takes an opinionated view of the application and configures various components automatically, by registering beans based on various criteria. The criteria can be:

* Availability of a particular class in a classpath
* Presence or absence of a Spring bean
* Presence of a system property
* An absence of a configuration file

For example, if you have the spring-webmvc dependency in your classpath, Spring Boot assumes you are trying to build a SpringMVC-based web application and automatically tries to register DispatcherServlet if it is not already registered. If you have any embedded database drivers in the classpath, such as H2 or HSQL, and if you haven’t configured a DataSource bean explicitly, then Spring Boot will automatically register a DataSource bean using in-memory database settings.

You will learn more about the autoconfiguration on [**What is Spring Boot Auto Configuration?**](https://youtu.be/qm_lBjlwQLY)

**3.3. Elegant Configuration Management**

Spring supports externalizing configurable properties using the [**@PropertySource**](http://www.javaguides.net/2018/09/spring-propertysource-annotation-with-example.html) configuration. Spring Boot takes it even further by using the sensible defaults and powerful type-safe property binding to bean properties. Spring Boot supports having separate configuration files for different profiles without requiring many configurations.

Read more [**http://www.javaguides.net/2018/09/spring-propertysource-annotation-with-example.html**](http://www.javaguides.net/2018/09/spring-propertysource-annotation-with-example.html)

**3.4. Spring Boot Actuator**

Being able to get the various details of an application running in production is crucial to many applications. The Spring Boot actuator provides a wide variety of such production-ready features without requiring developers to write much code. Some of the Spring actuator features are:

* Can view the application bean configuration details
* Can view the application URL mappings, environment details, and configuration parameter values
* Can view the registered health check metrics

Read more about Spring Boot Actuator on [**Spring Boot Actuator**](https://www.baeldung.com/spring-boot-actuators)

**3.5. Easy-to-Use Embedded Servlet Container Support**

Traditionally, while building web applications, you need to create WAR type modules and then deploy them on external servers like Tomcat, WildFly, etc. But by using Spring Boot, you can create a JAR type module and embed the servlet container in the application very easily so that the application will be a self-contained deployment unit.

Also, during development, you can easily run the Spring Boot JAR type module as a Java application from the IDE or from the command-line using a build tool like [**Maven**](http://www.javaguides.net/p/maven.html) or Gradle.

**4. Different Ways to Create Spring Boot Project**

There are three simplest ways to create a Spring boot project:

**1. Using spring initializ**r and import in STS, Eclipse, IntelliJ idea, VSCode, Netbeans

**2. Using Spring Starter Project** in STS (Eclipse)

**3. Spring Boot CLI** - The Spring Boot CLI is a command-line tool that you can use if you want to quickly develop a Spring application.

**5. Develop Spring Boot Application step by step and build few REST APIs**

Next, let's create a Spring boot application step by step and build few REST APIs.

**1. Creating spring boot application using spring initializr**

Spring Boot provides a web tool called [**https://start.spring.io**](https://start.spring.io/) to bootstrap an application quickly. Just go to [**https://start.spring.io**](https://start.spring.io/) and generate a new spring boot project.

**Use the below details in the Spring boot creation:**

**Project Name:** springboot-first-app

**Project Type:**Maven

**Choose dependencies:** Spring Web

**Package name:**com.springboot.app

**2. Maven Dependencies**

Refer below complete *pom.xml* 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>2.5.0</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<groupId>com.springboot.app</groupId>

<artifactId>springboot-first-app</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>springboot-first-app</name>

<description>Spring Boot First Application</description>

<properties>

<java.version>16</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>

</dependencies>

<build>

<plugins>

<plugin>

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

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

</plugin>

</plugins>

</build>

</project>

**3. Spring Boot Hello World REST API**

Let's create a HelloWorldController class and the below code to it:

package com.springboot.first.app;

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

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

@RestController

public class HelloWorldController {

// GET HTTP Method

// http://localhost:8080/hello-world

@GetMapping("/hello-world")

public String helloWorld() {

return "Hello World!";

}

}

* The above code uses Spring 4’s new @RestController annotation, which marks the class as a controller where every method returns a domain object instead of a view. It’s shorthand for @Controller and @ResponseBody rolled together.
* **@GetMapping** annotation for mapping HTTP GET requests onto specific handler methods. Specifically, **@GetMapping** is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.GET).

**Run Spring Boot Application:**

The below class SpringbootFirstAppApplication is the entry point that sets up the Spring Boot application. The @SpringBootApplication annotation enables auto-configuration and component scanning.

package com.springboot.first.app;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class SpringbootFirstAppApplication {

public static void main(String[] args) {

SpringApplication.run(SpringbootFirstAppApplication.class, args);

}

}

**Run spring boot application from the IDE:**

From your IDE, run the SpringbootFirstAppApplication.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/**](http://localhost:8080/).

**Run spring boot application using the command line:**

Just go to the root directory of the application and type the following command to run it -

$ mvn spring-boot:run

The application will start at Spring Boot’s default tomcat port **8080**.

Just hit this link in a browser: [**http://localhost:8080/hello-world**](http://localhost:8080/hello-world). You will able to see the response of this REST API in the browser.

**4. Build Spring Boot REST API returns Java Bean**

In this section, we will build a simple Spring boot REST API which returns Java bean as JSON to the client.

Let's first create a Student java bean class that the REST API want to return to the client:

package com.springboot.first.app;

public class Student {

private String firstName;

private String lastName;

public Student(String firstName, String lastName) {

super();

this.firstName = firstName;

this.lastName = lastName;

}

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;

}

}

Let's create the StudentController class and the below code to it:

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

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

@RestController

public class StudentController {

// http://localhost:8080/student

@GetMapping("/student")

public Student getStudent() {

return new Student("Ramesh", "Fadatare");

}

}

* The above code uses Spring 4’s new @RestController annotation, which marks the class as a controller where every method returns a domain object instead of a view. It’s shorthand for @Controller and @ResponseBody rolled together.
* **@GetMapping** annotation for mapping HTTP GET requests onto specific handler methods. Specifically, **@GetMapping** is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.GET).
* The Student object must be converted to JSON. Thanks to Spring’s HTTP message converter support, you don’t need to do this conversion manually. Because Jackson 2 is on the classpath, Spring’s MappingJackson2HttpMessageConverter is automatically chosen to convert the Student instance to JSON.

**Run spring boot application from the IDE:**

package com.springboot.first.app;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class SpringbootFirstAppApplication {

public static void main(String[] args) {

SpringApplication.run(SpringbootFirstAppApplication.class, args);

}

}

From your IDE, run the SpringbootFirstAppApplication.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/**](http://localhost:8080/).

Just hit this link in a browser: <http://localhost:8080/student>. You will able to see the response of this REST API in the browser.

**5. Build Spring Boot REST API returns List**

In this section, we will build a simple Spring boot REST API which returns List object as JSON to the client.

Let's open the StudentController class and the below code to it:

import java.util.ArrayList;

import java.util.List;

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

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

@RestController

public class StudentController {

@GetMapping("/students")

public List<Student> getStudents(){

List<Student> students = new ArrayList<>();

students.add(new Student("Ramesh", "Fadatare"));

students.add(new Student("Tony", "Cena"));

students.add(new Student("Sanjay", "Jadhav"));

students.add(new Student("Ram", "Jadhav"));

students.add(new Student("Umesh", "Fadatare"));

return students;

}

}

* The above code uses Spring 4’s new @RestController annotation, which marks the class as a controller where every method returns a domain object instead of a view. It’s shorthand for @Controller and @ResponseBody rolled together.
* **@GetMapping** annotation for mapping HTTP GET requests onto specific handler methods. Specifically, **@GetMapping** is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.GET).
* The List of student objects must be converted to JSON. Thanks to Spring’s HTTP message converter support, you don’t need to do this conversion manually. Because Jackson 2 is on the classpath, Spring’s MappingJackson2HttpMessageConverter is automatically chosen to convert the List of student objects to JSON.

**Run spring boot application from the IDE:**

From your IDE, run the SpringbootFirstAppApplication.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/**](http://localhost:8080/).

Just hit this link in a browser: [**http://localhost:8080/students**](http://localhost:8080/students). You will able to see the response of this REST API in the browser.

**6. Spring Boot REST API with Path Variable**

In this section, we will build a simple Spring boot REST API to demonstrate the usage of *@PathVariable* annotation.

Let's open the StudentController class and the below code to it:

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

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

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

@RestController

public class StudentController {

// http://localhost:8080/student/1

// @PathVariable annotation

@GetMapping("/student/{firstName}/{lastName}/")

public Student studentPathVariable(@PathVariable("firstName") String firstName,

@PathVariable("lastName") String lastName) {

return new Student(firstName, lastName);

}

}

* The above code uses Spring 4’s new @RestController annotation, which marks the class as a controller where every method returns a domain object instead of a view. It’s shorthand for @Controller and @ResponseBody rolled together.
* **@GetMapping** annotation for mapping HTTP GET requests onto specific handler methods. Specifically, **@GetMapping** is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.GET).
* The Student object must be converted to JSON. Thanks to Spring’s HTTP message converter support, you don’t need to do this conversion manually. Because Jackson 2 is on the classpath, Spring’s MappingJackson2HttpMessageConverter is automatically chosen to convert the Student object to JSON.
* With the @PathVariable annotation, we bind the request URL template path variable to the method variable. For instance, with the **/Ramesh/Fadatare/** URL, the **"Ramesh"**value is bound to the **firstName**variable and the "**Fadatare**" value to the **lastName**variable.

**Run spring boot application from the IDE:**

From your IDE, run the SpringbootFirstAppApplication.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/**](http://localhost:8080/).

Just hit this link in a browser: [**http://localhost:8080/student/Ramesh/Fadatare**](http://localhost:8080/student/Ramesh/Fadatare). You will be able to see the response of this REST API in the browser.

**7. Build Spring Boot REST API with Request Param**

In this section, we will build a simple Spring boot REST API that handles request or query parameters in the GET HTTP request.

In this example, we use the @RequestParam annotation to extract query parameters from the HTTP GET request.

Let's open the StudentController class and the below code to it:

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

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

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

@RestController

public class StudentController {

// build rest API to handle query parameters

// http://localhost:8080/student/query?firstName=Ramesh&lastName=Fadatare

@GetMapping("/student/query")

public Student studentQueryParam(

@RequestParam(name = "firstName") String firstName,

@RequestParam(name = "lastName") String lastName) {

return new Student(firstName, lastName);

}

}

* The above code uses Spring 4’s new @RestController annotation, which marks the class as a controller where every method returns a domain object instead of a view. It’s shorthand for @Controller and @ResponseBody rolled together.
* **@GetMapping** annotation for mapping HTTP GET requests onto specific handler methods. Specifically, **@GetMapping** is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.GET).
* The Student object must be converted to JSON. Thanks to Spring’s HTTP message converter support, you don’t need to do this conversion manually. Because Jackson 2 is on the classpath, Spring’s MappingJackson2HttpMessageConverter is automatically chosen to convert the Student object to JSON.
* With the *@RequestParam* annotation, we extract query parameters from the HTTP GET request.

**6. Develop Spring Boot CRUD REST APIs with Hibernate and MySQL Database**

In this section, you will learn how to build REST APIs using the latest version of Spring boot, hibernate, and MySQL database.

**Tools and technologies used:**

* Java 16
* Spring Boot
* Spring Data JPA ( Hibernate)
* Lombok
* MySQL
* Eclipse STS
* Maven
* Tomcat
* Postman

Let's create a Spring boot application step by step to build CRUD REST APIs.

**1. Create Spring Boot Project**

Spring Boot provides a web tool called [**Spring Initializer**](https://start.spring.io/) to bootstrap an application quickly. Just go to [**https://start.spring.io/**](https://start.spring.io/) and generate a new spring boot project.

**Use the below details in the Spring boot creation:**

**Project Name:** springboot-backend

**Project Type:**Maven

**Choose dependencies:** Spring Web, Spring Data JPA, MySQL Driver, Lombok

**Package name:**net.javaguides.springboot

**2. Maven Dependencies**

<?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>2.5.0</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<groupId>net.javaguides</groupId>

<artifactId>springboot-backend</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>springboot-backend</name>

<description>Spring Boot RESTful Web Services</description>

<properties>

<java.version>16</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>

</project>

**3. Spring Boot Project Architecture**

We are going to use three-layer architecture in our Spring boot project:

**[Diagram

Description automatically generated](https://1.bp.blogspot.com/-8d817gzzfEA/YNbqNJxpgOI/AAAAAAAAI2I/V3EX4_rqInkX3i0IF9CG7EhlPvXG-Qy0ACLcBGAsYHQ/s1280/Spring%2BBoot%2BProject%2BStructure.PNG)**

**4. Configure MySQL Database**

Spring Boot tries to auto-configure a DataSource if spring-data-jpa dependency is in the classpath by reading the database configuration from the application.properties file.

So, we just have to add the configuration, and Spring Boot will take care of the rest.

Open the application.properties file and add the following properties to it.

## Spring DATASOURCE (DataSourceAutoConfiguration & DataSourceProperties)

spring.datasource.url = jdbc:mysql://localhost:3306/demo?useSSL=false&serverTimezone=UTC&useLegacyDatetimeCode=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

You will need to create a database named demo in MySQL, and change the spring.datasource.username & spring.datasource.password properties as per your MySQL installation.

In the above properties file, the last two properties are for Hibernate. **Spring Boot uses Hibernate as the default JPA implementation.**

The property spring.jpa.hibernate.ddl-auto is used for database initialization. I’ve used the value “update” for this property.

**5. Create JPA Entity - Employee**

Let's create a new package called model inside net.javaguides.springboot and add a class named Employee.java with the following contents:

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import javax.persistence.Table;

import lombok.Data;

@Data

@Entity

@Table(name="employees")

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private long id;

@Column(name = "first\_name", nullable = false)

private String firstName;

@Column(name = "last\_name")

private String lastName;

@Column(name = "email")

private String email;

}

Note that we are using Lombok annotation @Data to reduce the boilerplate code (getters/setters).

**6. Create Repository or DAO Layer - EmployeeRepository**

The next thing we’re gonna do is create a repository to access an Employee’s data from the database.

The JpaRepository interface defines methods for all the CRUD operations on the entity, and a default implementation of the JpaRepository called SimpleJpaRepository.

Let’s create the repository now. First, create a new package called repository inside the base package net.javaguides.springboot. Then, create an interface called EmployeeRepository and extend it from JpaRepository -

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

import net.javaguides.springboot.model.Employee;

public interface EmployeeRepository extends JpaRepository<Employee, Long>{

}

Note that we no need to add @Repository annotation because Spring Data JPA internally takes care of it.

**7. Create Service Layer**

Let's implement service layer, we will first create interface and then it's implementation class.

**7.1 Service Interface**

Let's create a package called service inside base package net.javaguides.springboot. Create an EmployeeService interface with the following contents:

import java.util.List;

import net.javaguides.springboot.model.Employee;

public interface EmployeeService {

Employee saveEmployee(Employee employee);

List<Employee> getAllEmployees();

Employee getEmployeeById(long id);

Employee updateEmployee(Employee employee, long id);

void deleteEmployee(long id);

}

**7.2 Service Interface Implementation**

Let's create a package called impl inside package net.javaguides.springboot.service. Create an EmployeeServiceImpl class with the following contents:

import java.util.List;

import java.util.Optional;

import org.springframework.stereotype.Service;

import net.javaguides.springboot.exception.ResourceNotFoundException;

import net.javaguides.springboot.model.Employee;

import net.javaguides.springboot.repository.EmployeeRepository;

import net.javaguides.springboot.service.EmployeeService;

@Service

public class EmployeeServiceImpl implements EmployeeService{

private EmployeeRepository employeeRepository;

public EmployeeServiceImpl(EmployeeRepository employeeRepository) {

super();

this.employeeRepository = employeeRepository;

}

@Override

public Employee saveEmployee(Employee employee) {

return employeeRepository.save(employee);

}

@Override

public List<Employee> getAllEmployees() {

return employeeRepository.findAll();

}

@Override

public Employee getEmployeeById(long id) {

// Optional<Employee> employee = employeeRepository.findById(id);

// if(employee.isPresent()) {

// return employee.get();

// }else {

// throw new ResourceNotFoundException("Employee", "Id", id);

// }

return employeeRepository.findById(id).orElseThrow(() ->

new ResourceNotFoundException("Employee", "Id", id));

}

@Override

public Employee updateEmployee(Employee employee, long id) {

// we need to check whether employee with given id is exist in DB or not

Employee existingEmployee = employeeRepository.findById(id).orElseThrow(

() -> new ResourceNotFoundException("Employee", "Id", id));

existingEmployee.setFirstName(employee.getFirstName());

existingEmployee.setLastName(employee.getLastName());

existingEmployee.setEmail(employee.getEmail());

// save existing employee to DB

employeeRepository.save(existingEmployee);

return existingEmployee;

}

@Override

public void deleteEmployee(long id) {

// check whether a employee exist in a DB or not

employeeRepository.findById(id).orElseThrow(() ->

new ResourceNotFoundException("Employee", "Id", id));

employeeRepository.deleteById(id);

}

}

**8. Create Custom Exception**

Let's create an exception package inside base package net.javaguides.springboot. Then create a ResourceNotFoundException custom exception and add the following contents to it:

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 = 1L;

private String resourceName;

private String fieldName;

private Object fieldValue;

public ResourceNotFoundException(String resourceName, String fieldName, Object fieldValue) {

super(String.format("%s not found with %s : '%s'", resourceName, fieldName, fieldValue));

this.resourceName = resourceName;

this.fieldName = fieldName;

this.fieldValue = fieldValue;

}

public String getResourceName() {

return resourceName;

}

public String getFieldName() {

return fieldName;

}

public Object getFieldValue() {

return fieldValue;

}

}

**9. Create Controller Layer**

We’ll now create the REST APIs for creating, retrieving, updating, and deleting an Employee.

First, create a new package controller inside the base package net.javaguides.springboot. Then, create a new class EmployeeController.java with the following contents -

import java.util.List;

import org.springframework.http.HttpStatus;

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.model.Employee;

import net.javaguides.springboot.service.EmployeeService;

@RestController

@RequestMapping("/api/employees")

public class EmployeeController {

private EmployeeService employeeService;

public EmployeeController(EmployeeService employeeService) {

super();

this.employeeService = employeeService;

}

// build create employee REST API

@PostMapping()

public ResponseEntity<Employee> saveEmployee(@RequestBody Employee employee){

return new ResponseEntity<Employee>(employeeService.saveEmployee(employee), HttpStatus.CREATED);

}

// build get all employees REST API

@GetMapping

public List<Employee> getAllEmployees(){

return employeeService.getAllEmployees();

}

// build get employee by id REST API

// http://localhost:8080/api/employees/1

@GetMapping("{id}")

public ResponseEntity<Employee> getEmployeeById(@PathVariable("id") long employeeId){

return new ResponseEntity<Employee>(employeeService.getEmployeeById(employeeId), HttpStatus.OK);

}

// build update employee REST API

// http://localhost:8080/api/employees/1

@PutMapping("{id}")

public ResponseEntity<Employee> updateEmployee(@PathVariable("id") long id

,@RequestBody Employee employee){

return new ResponseEntity<Employee>(employeeService.updateEmployee(employee, id), HttpStatus.OK);

}

// build delete employee REST API

// http://localhost:8080/api/employees/1

@DeleteMapping("{id}")

public ResponseEntity<String> deleteEmployee(@PathVariable("id") long id){

// delete employee from DB

employeeService.deleteEmployee(id);

return new ResponseEntity<String>("Employee deleted successfully!.", HttpStatus.OK);

}

}

**10. Run Spring Boot Application**

We’ve successfully built all the APIs for our application. Let’s now run the app and test the APIs.

Just go to the root directory of the application and type the following command to run it -

$ mvn spring-boot:run

The application will start at Spring Boot’s default tomcat port **8080**.