Spring Day3

Spring Boot:

- Spring Boot is a project that is built on the top of the Spring Framework. It provides an easier and faster way to set up, configure, and run both simple and web-based applications.
- It is a spring platform that mainly focuses on convention over configuration.
- Spring boot is not a framework software, it is a just spring application development platform that provides RAD(rapid application development) features to the spring framework.
- With the spring boot, our spring application development process will be simplified.
- Spring boot supports auto-configurations
- It provides an integrated tomcat server, so we can develop a **Web-based applications** without installing the Webserver separately.
- With the help of spring boot, we can develop **Web-application, web services, microservices, and spring cloud-based applications** very easily.

There are various ways to use spring boot to develop a spring application:

1. Installing STS software
2. installing the STS plugin in normal eclipse software
3. By using the Spring initializer website
4. By using Maven or Gradle application.
5. By using Spring Boot CLI (Command Line interface)

Developing a Spring Boot starter project using STS software (Standalone or Simple Spring application):

Step 1: create a new spring starter project in the STS software:

File → New → Spring Starter Project → provide the Name (App1), group, artifact of the project → next → for a simple application, don't select any dependencies →next → finish.

The above step will generate a basic spring boot application (a maven project) with all the initial configurations.

The above project skeleton will have a spring boot configuration file **applicaiton.properties** inside the **src/main/resources** folder.

and one Main launcher class(App1Application) inside the **src/main/java** folder.

The Main Launcher class is annotated with **@SpringBootApplication** annotation.

Example:

```
package com.masai;

import org.springframework.boot.SpringApplication;
```

```
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class App1Application {

  public static void main(String[] args) {
    SpringApplication.run(App1Application.class, args);
  }

}
```

This @**SpringBootApplication** annotation is a combination of:

@Configuration: (This annotation marks a class as a Configuration class for Java-based configuration)
Spring Boot auto-configuration attempts to automatically configure your Spring application based on the jar dependencies that you have added
. For example, If HSQLDB is on your classpath, and you have not manually configured any database connection beans, then we will auto-configure an in-memory database.

@ComponentScan: (here the base package is in which package this class belongs)

@EnableAutoConfiguration: (to enable Spring Boot's auto-configuration feature)

In the above application **SpringApplication.run(App1Application.class, args);** will return **ApplicationContext** container, we can pull any spring bean object and can call the business method on it.

Example:

Develop another class A.java inside the same package and register this class with the Spring container by applying a stereotype annotation on it.

```
A.java:
-------

package com,masai;

import org.springframework.stereotype.Service;

@Service
public class A {

  public void funA() {
    System.out.println("inside funA of A");
  }
}
```

Since the above class A is annotated with @Service annotation, it will be automatically registered with the spring container with the id **'a'** :

Example: Inside the main application we can pull the A class object and can call funA() method.

```
package com.masai;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;

@SpringBootApplication
public class App1Application {
```

```
  public static void main(String[] args) {
    ApplicationContext ctx= SpringApplication.run(App1Application.class, args);

    A a1= ctx.getBean("a", A.class);

    a1.funA();

  }

}
```

Spring vs. Spring Boot:

Spring: Spring Framework is the most popular application development framework of Java. The main feature of the Spring Framework is **dependency Injection** or **Inversion of Control** (IoC). With the help of Spring Framework, we can develop a **loosely** coupled application. It is better to use if application type or characteristics are purely defined.

Spring Boot: Spring Boot is a module of Spring Framework. It allows us to build a stand-alone application with minimal or zero configurations. It is better to use if we want to develop a simple Spring-based application or RESTful services.

The primary comparison between Spring and Spring Boot are discussed below:

Spring	Spring Boot
Spring Framework is a widely used Java EE framework for building applications.	**Spring Boot Framework** is widely used to develop **REST APIs** .
It aims to simplify Java EE development that makes developers more productive.	It aims to shorten the code length and provide the easiest way to develop **Web Applications** .
The primary feature of the Spring Framework is **dependency injection** .	The primary feature of Spring Boot is **Autoconfiguration** . It automatically configures the classes based on the requirement.
It helps to make things simpler by allowing us to develop **loosely coupled** applications.	It helps to create a **stand-alone** application with less configuration.
The developer writes a lot of code (**boilerplate code**) to do the minimal task.	It **reduces** boilerplate code.
To test the Spring project, we need to set up the sever explicitly.	Spring Boot offers **embedded server** such as **Jetty** and **Tomcat** , etc.
It does not provide support for an in-memory database.	It offers several plugins for working with an embedded and **in-memory** database such as **H2** .
Developers manually define dependencies for the Spring project in **pom.xml** .	Spring Boot comes with the concept of **starter** in pom.xml file that internally takes care of downloading the dependencies **JARs** based on Spring Boot Requirement.

Spring Boot vs. Spring MVC:

Spring Boot: Spring Boot makes it easy to quickly bootstrap and start developing a Spring-based application. It avoids a lot of boilerplate code. It hides a lot of complexity behind the scene so that the developer can quickly get started and develop Spring-based applications easily.

Spring MVC: Spring MVC is a Web MVC Framework for building web applications. It contains a lot of configuration files for various capabilities. It is an HTTP oriented web application development framework.

Spring Boot and Spring MVC exist for different purposes. The primary comparison between Spring Boot and Spring MVC are discussed below:

Spring Boot	Spring MVC
Spring Boot is a module of Spring for packaging the Spring-based application with sensible defaults.	**Spring MVC** is a model view controller-based web framework under the Spring framework.
It provides default configurations to build **Spring-powered** framework.	It provides **ready to use** features for building a web application.
There is no need to build configuration manually.	It requires build configuration manually.
There is **no requirement** for a deployment descriptor.	A Deployment descriptor is **required**.
It avoids boilerplate code and wraps dependencies together in a single unit.	It specifies each dependency separately.
It **reduces** development time and increases productivity.	It takes **more** time to achieve the same.

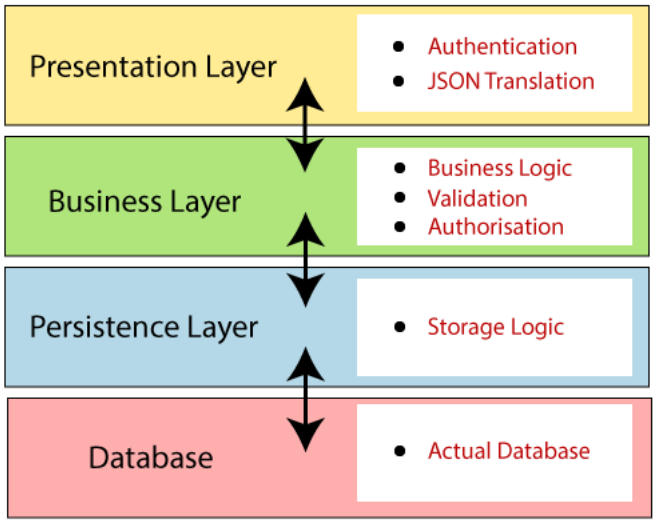
Spring Boot Architecture:

Spring Boot is a module of the Spring Framework. It is used to create stand-alone, production-grade Spring Based Applications with minimum efforts. It is developed on top of the core Spring Framework.

Spring Boot follows a layered architecture in which each layer communicates with the layer directly below or above (hierarchical structure) it.

Before understanding the **Spring Boot Architecture**, we must know the different layers and classes present in it. There are **four** layers in Spring Boot are as follows:

- **Presentation Layer**
- **Business Layer**
- **Persistence Layer**
- **Database Layer**



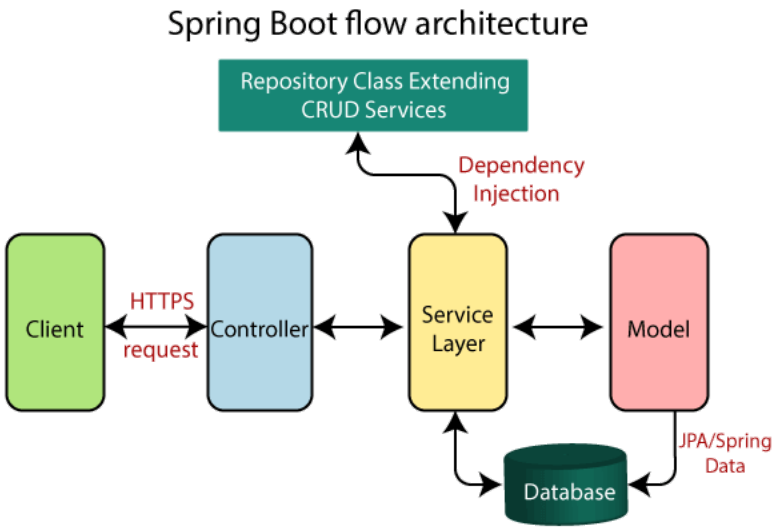
Presentation Layer: The presentation layer handles the HTTP requests, translates the JSON parameter to object, and authenticates the request and transfer it to the business layer. In short, it consists of **views** i.e., frontend part.

Business Layer: The business layer handles all the **business logic**. It consists of service classes and uses services provided by data access layers. It also performs **authorization** and **validation**.

Persistence Layer: The persistence layer contains all the **storage logic** and translates business objects from and to database rows.

Database Layer: In the database layer, **CRUD** (create, retrieve, update, delete) operations are performed.

Spring Boot Flow Architecture:



Web Services:

A java service that can be accessed over the web is known as Webservice.

Web-Application can also be accessed over the web but there are some differences:

1. Web-Application is meant for the end-user and to be accessed by the browser in a human-readable format. whereas a Web Service is meant for applications to access the raw data in the form of XML, JSON, plain text, etc.
2. A Web-Application can access the Web-Services to access some data or to perform some tasks. whereas a Web Service can not access a Web Application to get some data.
3. A single Web-Service can be used by different kinds of applications (developed in any technology like Java, .Net, Angular, React, Phyton, etc.).
   whereas a Web Application is not meant for reusability.

Note:- One application can communicate with another application using Web Services, and both applications can be developed in any technology, Java, .Net, Angular, React, Phyton).

Web Services are categorized into 2 types:

1. **SOAP-based Webservices**
2. **Restful Webservices**

SOAP-based WS:- (Simple object access protocol)

It is an XML based protocol, to communicate between the client applications to the server application, since it is XML based, it is platform-independent and language independent.

To represent the Student object in xml:

```
<student>
<roll>100</roll>
<name>ram</name>
<marks>100</marks>
</student>
```

Creating a Web service using SOAP are heavyweight application because apart from creating the main service classes (actual service) there is a need to create some extra binding classes to parse the XML and convert them in Java objects and vice-versa moreover to expose and consume the SOAP-based web services we need to generate some extra xml file (WSDL file).

In order to make web services as light-weight, another architectural style is defined named as **"REST" (Representational State transfer).**

This Rest architectural style tells that create a web service with the required operation and expose them through the **HTTP protocol**. and allow them to be accessed by the client with the help of http protocol only without any binding classes and any XML files.

Based on this architectural style sun-microsystem has released **JAX-RS** API for creating
Restful web-services in java.

It does not define the standard message exchange format. We can build REST services with both XML and JSON. JSON is the more popular format with REST. The **key abstraction** is a resource in REST. A resource can be anything. It can be accessed through a **Uniform Resource Identifier (URI)**. For example:

The resource has representations like XML, HTML, and JSON. The current state is capture by representational resources. When we request a resource, we provide the representation of the resource. The important methods of HTTP are:

- **GET:** It reads a resource.
- **PUT:** It updates an existing resource.
- **POST:** It creates a new resource.
- **DELETE:** It deletes the resource.

For example, if we want to perform the following actions in the social media application, we get the corresponding results.

POST /users: It creates a user.

GET /users/{id}: It retrieves the detail of a user.

GET /users: It retrieves the detail of all users.

DELETE /users: It deletes all users.

DELETE /users/{id}: It deletes a user.

GET /users/{id}/posts/post_id: It retrieve the detail of a specific post.

POST / users/{id}/ posts: It creates a post of the user.

HTTP also defines the following standard status code:

- **404:** RESOURCE NOT FOUND
- **200:** SUCCESS
- **201:** CREATED
- **401:** UNAUTHORIZED
- **500:** SERVER ERROR

Advantages of RESTful web services

- RESTful web services are **platform-independent**.
- It can be written in any programming language and can be executed on any platform.
- It provides different data formats like **JSON, text, HTML,** and **XML**.
- It is fast in comparison to SOAP because there is no strict specification like SOAP.
- These are **reusable**.
- They are **language neutral**.

Spring framework itself has provided its own API with the implementation to make a Controller bean (Controller bean is the part of Spring-MVC based web application). which is annotated with **@Controller** as a Restful web service class.

Converting Spring MVC controller bean as a Restful service:

@Controller + @ResponseBody = @RestController

Steps to create the first Spring boot REST web service :

Step 1: Create a spring starter project on STS.

Step 2: select the spring web dependency, (This dependency will include an integrated Tomcat web server, in which our application will be deployed)

Step 3: Creating a HelloWorld service:

Create a new class with the name **HelloWorldController** inside the main package(com.masai).

To make this class a Restful web service class, annotate this class with the following annotation:

@RestController

Step 4: Inside this class define the actual service (a method that can perform some operation)

we need to make this service (method) accessible with an URI by the help of **@RequestMapping annotation.** and we need to specify the appropriate HTTP method using which we want to expose this service.

Example :

```
package com.masai;

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

@RestController
public class HelloWorldController {

  @RequestMapping(value= "/hi",method = RequestMethod.GET)
  public String sayHello() {
    return "welcome"; // it will be represented as raw data.
  }
}
```

We can also improve the above code by using the **@GetMapping** annotation instead of @RequestMapping. Here the method specification is not required.

Example:

```
package com.masai;

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

@RestController
public class HelloWorldController {

  @GetMapping("/hi")
  public String sayHello() {
    return "welcome"; // it will be represented as raw data.
  }
}
```

Step 5: Change the port number of the Integrated Tomcat server (8080 is the most common port, it is recommended to change it).

inside the **application.properties** file of the **src/main/resources** folder make the following entry:

```
server.port = 8088
```

Step 6: run the main class of the spring boot project (in which main method is there) as a spring boot application.

Step 7: from the browser give the following request from the URL.

http://localhost:8088/hi

Enhancing the above Service to Return a Bean:

Create a Student.java file as java bean class

```
package com.masai;

public class Student {
```

```
  private Integer roll;
  private String name;
  private Integer marks;

  public Student() {
  }

  public Student(Integer roll, String name, Integer marks) {
    super();
    this.roll = roll;
    this.name = name;
    this.marks = marks;
  }

  public Integer getRoll() {
    return roll;
  }

  public void setRoll(Integer roll) {
    this.roll = roll;
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public Integer getMarks() {
    return marks;
  }

  public void setMarks(Integer marks) {
    this.marks = marks;
  }
}
```

Define a service that will return Student object in the form of JSON data.

```
package com.masai;

import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;
@RestController
public class HelloWorldController {

  @RequestMapping(value= "/hi",method = RequestMethod.GET)
  public String sayHello() {
    return "welcome"; // it will be represented as raw data.
  }

  @GetMapping(value = "/student", produces = MediaType.APPLICATION_JSON_VALUE)
  public Student getStudent() {
    Student student = new Student();
    student.setRoll(10);
    student.setName("Ram");
    student.setMarks(850);

    return student;
  }

}
```

Run the application and give the request from the browser:

http://localhost:8088/student

Output:

```
{"roll":10,"name":"Ram","marks":850}
```

Note: while returning the response from the webservice methods spring f/w uses some of the **"message converters"** to convert the Java object into the domain format like JSON object, XML format, etc.

The default conversion type is JSON type, here spring f/w uses "message converters" with the help of **Jackson API** internally. so defining **MediaType.APPLICATION_JSON_VALUE** is optional.

Exmaple:

```
@GetMapping(value = "/student")
  public Student getStudent() {
    Student student = new Student();
    student.setRoll(10);
    student.setName("Ram");
    student.setMarks(850);

    return student;
  }

}
```

Defining a service that return List of Student:

```
@GetMapping(value = "/students")
  public List<Student> getAllStudents() {

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

    students.add(new Student(10, "Ram", 850));
    students.add(new Student(12, "Ramesh", 650));
    students.add(new Student(13, "Ravi", 750));
    students.add(new Student(14, "amit", 950));

    return students;

  }
```

http://localhost:8088/students

Output:

```
[
  {"roll":10,"name":"Ram","marks":850},
  {"roll":12,"name":"Ramesh","marks":650},
  {"roll":13,"name":"Ravi","marks":750},
  {"roll":14,"name":"amit","marks":950}
]
```

Note: The above mapping information we can provide by following ways, all the same:

```
  1. @RequestMapping("/students")
  2. @RequestMapping(value = "/students", method = RequestMethod.GET)
  3. @GetMapping("/studsnts") // it is the sortcut
  4. @RequestMapping(value = "/students",method = RequestMethod.GET,produces = "application/json")
  5. @RequestMapping(value = "/students",method = RequestMethod.GET,produces =MediaType.APPLICATION_JSON_VALUE)
```

Note: After every change, we need to restart the server to deploy the application. If we want every change will automatically redeploy the application then we need to add the following dependency inside the pom.xml:

```
<!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-devtools -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <version>2.6.6</version>
</dependency>
```

Note: @RequestMapping annotation we can apply to the root resource (to the Controller class) also.

Exmaple:

```
@RestController
@RequestMapping("/studentapp")   // to the root resource
public class HelloWorldController {

  @GetMapping(value = "/student")
  public Student getStudent() {
    Student student = new Student();
    student.setRoll(10);
    student.setName("Ram");
    student.setMarks(850);

    return student;
  }

}
```

Now we need to give the request to our service as follows:

```
http://localhost:8088/studentapp/student
```

@PathVariable:

The **@PathVariable** annotation is used to extract the value from the URI. It is most suitable for the RESTful web service where the URL contains some value. Spring MVC allows us to use multiple @PathVariable annotations in the same method. A path variable is a critical part of creating rest resources.

```
  @GetMapping(value = "/student/{roll}")
    public Student getStudent(@PathVariable("roll") Integer roll) {
      Student student = new Student();
      student.setRoll(roll);  // using path variable
      student.setName("Ram");
      student.setMarks(850);

      return student;
    }
```

```
  }
```

Now we need to give the request to our service with the path variable:

Example:

```
http://localhost:8088/studentapp/student/10
```

Defining multiple PathVariable:

```
package com.masai;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
@RestController
@RequestMapping("/studentapp")
public class HelloWorldController {

  @GetMapping(value = "/student/{r}/{n}/{m}")
  public Student getStudent(@PathVariable("r") Integer roll,
               @PathVariable("n") String name,
               @PathVariable("m") Integer marks) {
    Student student = new Student();
    student.setRoll(roll);  // using path variable
    student.setName(name);  // using path variable
    student.setMarks(marks); // using path variable

    return student;
  }

}
```

Give the request as follows:

http://localhost:8088/studentapp/student/10/Ram/500

@RequestParam:

This annotation is used to fetch query parameter/request parameter from the request.

Example:

```
http://localhost:8088/studentapp/student?name=Ram

If more than one parameter we need to pass then they need to be concataneted with & symbol
```

Example:

```
@GetMapping("/getStudent")
public Student getStudentDetails(@RequestParam Integer roll,@RequestParam String name,@RequestParam Integer marks) {

    return new Student(roll, name, marks);
  }
```

http://localhost:7000/studentApp/getStudent?roll=100&name=Ram&marks=500

Note:- By default @RequestParam is mandatory, to make it optional :

```
  @GetMapping("/getStudent")
  public Student getStudentDetails(@RequestParam(required = false) Integer roll,@RequestParam(required = false) String name,@RequestParam(r

    return new Student(roll, name, marks);
  }
```

Implementing Post method:

To implement the Post method, we need to use **@RequestBody** annotation.

This Post method is used to send some data (object) to our web-service method.

The @RequestBody annotation maps body of the web request to the method parameter. The body of the request is passed through an HttpMessageConverter. It resolves the method argument depending on the content type of the request.

Example:

```
@PostMapping(value = "/saveStudent",consumes = "application/json")
  public String saveStudentDetails(@RequestBody Student student) {

    //here we can communicate with the Service layer or Data Access Layer classes to
    //persist the Student object in the Database.

    return  "Student stored ,"+student.getName();

  }
```

Here By default consume type is JSON only, so **consumes = "application/json"** is optional.

Example:

```
@PostMapping(value = "/saveStudent")
  public String saveStudentDetails(@RequestBody Student student) {

    //here we can communicate with the Service layer or Data Access Layer classes to
    //persist the Student object in the Database.

    return  "Student stored ,"+student.getName();

  }
```

Note: From the browser, we can send only the GET request, to send the POST request along with data we need to use **REST client,** like Postman software.

Download the Postman from https://www.getpostman.com/downloads/

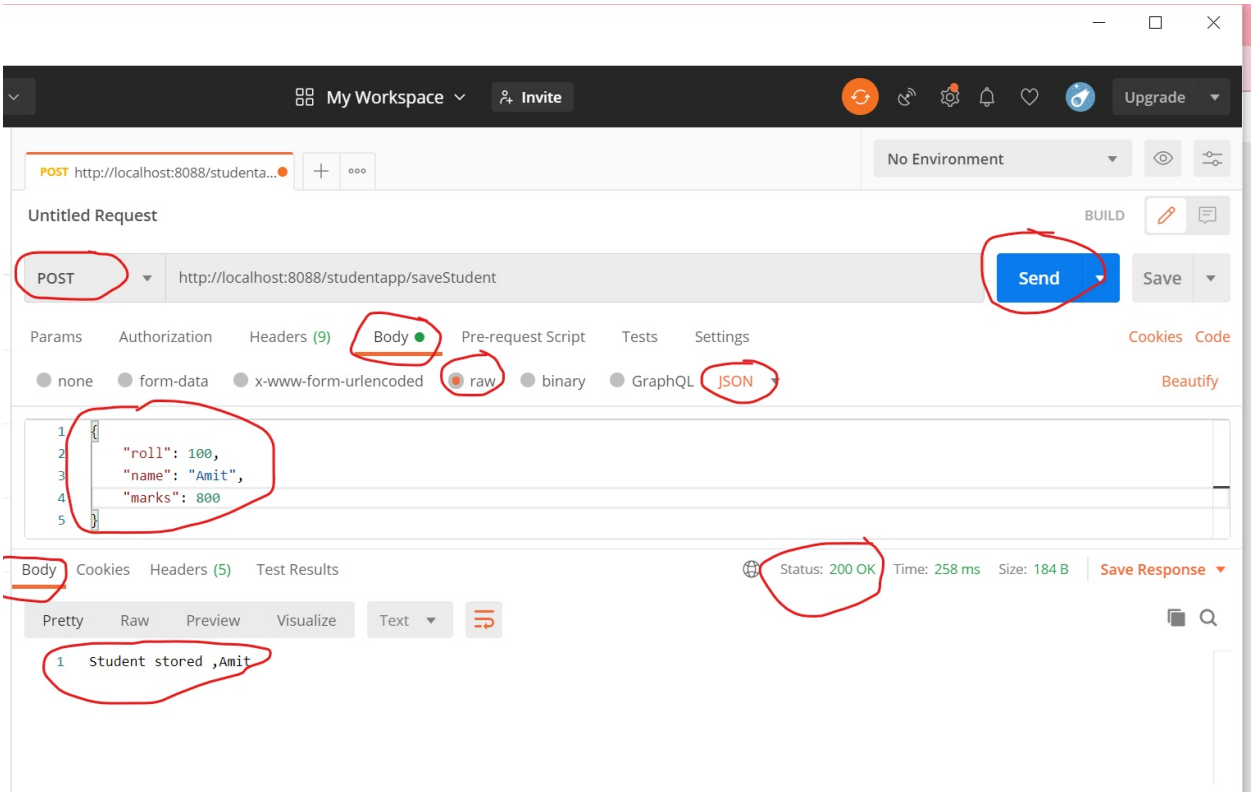
after installation, Launch the **Postman** and **Signup.**

From the Postman we need to send the POST request along with the Student object in the form of JSON object by using HTTP request Body.

Student Object in the form of JSON:

```
{
    "roll": 50,
    "name": "Ratan",
    "marks": 750
}
```

Note: Here key should be in double quote.



Implementing PUT method:

HTTP PUT method we use to update an existing resource:

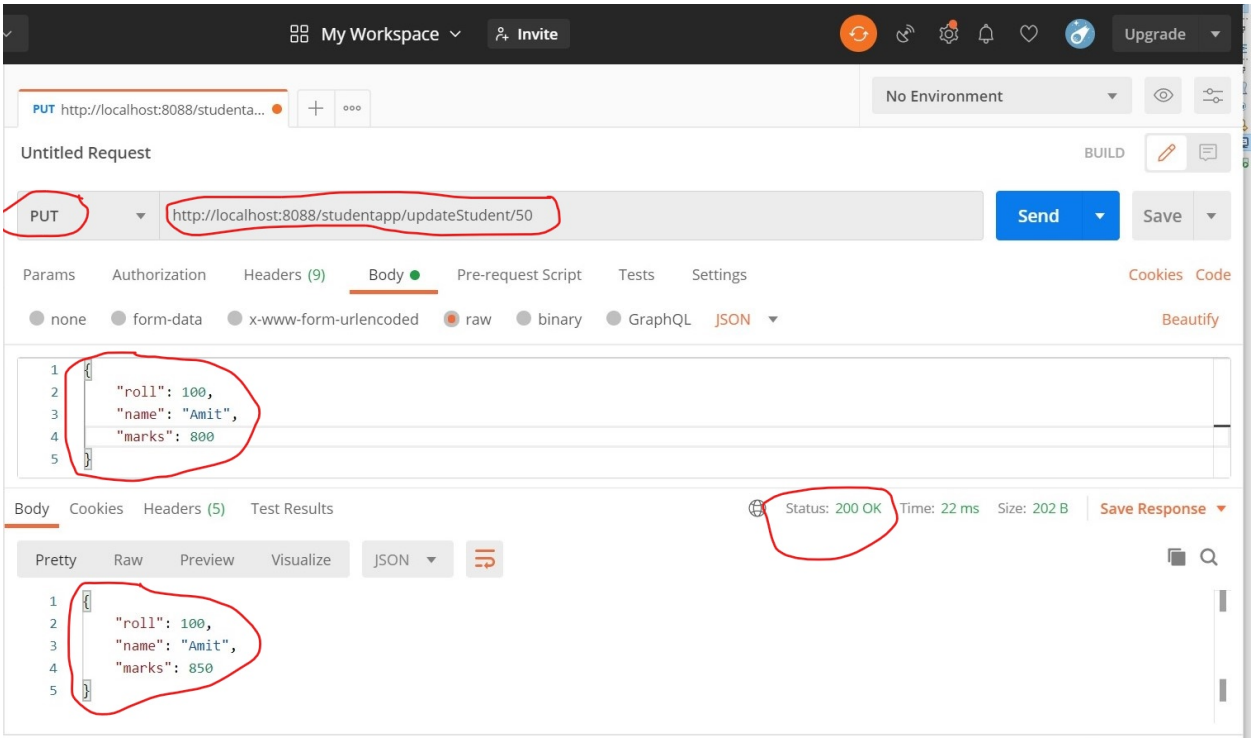
Example:

```
@PutMapping(value = "/updateStudent/{graceMarks}")
  public Student updateStudentDetails(@RequestBody Student student,@PathVariable("graceMarks")int gMarks) {

    student.setMarks(student.getMarks()+gMarks);

    return student;
  }
```

.
.



ResponseEntity:

ResponseEntity **represents the whole HTTP response: status code, headers, and body**. As a result, we can use it to fully configure the HTTP response.

If we want to use it, we have to return it from the endpoint; Spring takes care of the rest.

ResponseEntity is a generic type. Consequently, we can use any type as the response body:

Example:

```
@GetMapping("/hello")
public ResponseEntity<String> hello() {
    return new ResponseEntity<>("Hello World!", HttpStatus.OK);
}
```

Since we specify the response status programmatically, we can return with different status codes for different scenarios:

Example:

```
@GetMapping("/getAge/{age}")
public ResponseEntity<String> getAgeHandler(
  @PathVariable("age") int yearOfBirth) {

    if (isInFuture(yearOfBirth)) {
        return new ResponseEntity<>(
          "Year of birth cannot be in the future",
          HttpStatus.BAD_REQUEST);
```

```
    }

    return new ResponseEntity<>(
      "Your age is " + calculateAge(yearOfBirth),
      HttpStatus.OK);
}
```

Additionally, we can set HTTP headers: (Http Headers are some extra information about the response to the client).

```
  @PutMapping(value = "/updateStudent/{graceMarks}")
  public ResponseEntity<Student> updateStudentDetails(@RequestBody Student student,@PathVariable               ("graceMarks")int gMarks) {

    student.setMarks(student.getMarks()+gMarks);

    HttpHeaders hh=new HttpHeaders();

    hh.add("jwt", "sdnjnwjk3kj412321231sdf");
    hh.add("user", "admin");
    hh.add("hello", "abc");

    ResponseEntity<Student> re=new ResponseEntity<>(student,hh,HttpStatus.ACCEPTED);

    return re;
  }
```

References:
https://docs.spring.io/spring-boot/docs/1.3.8.RELEASE/reference/html/using-boot-auto-configuration.html#:~:text=Spring Boot auto-configuration attempts,configure an in-memory database.