

What is Spring Boot?

Spring Boot is a framework built on top of the **Spring Framework** that simplifies the development of Java-based applications. It **eliminates the need for extensive configuration** and allows developers to quickly build production-ready applications with minimal effort.

Spring Boot is an open-source Java framework built on top of the **Spring Framework**. It simplifies Java application development by providing:

- Auto-configuration** (reduces boilerplate code)
- Standalone applications** (no need for external servers like Tomcat)
- Embedded servers** (comes with Tomcat, Jetty, or Undertow)
- Production-ready features** (monitoring, logging, security)

It follows a "convention over configuration" approach, reducing manual setup and enabling developers to **build applications faster**.

Why are we moving to Spring Boot?

Traditional Java-based enterprise applications using **Spring Framework** require a lot of manual **configurations, XML setups, and dependency management**. Spring Boot removes this complexity by offering **auto-configuration, embedded servers, and simplified deployment**.

Advantages of Spring Boot

1. **Auto-Configuration:** Automatically configures components based on classpath dependencies.
2. **Embedded Servers:** Comes with built-in Tomcat, Jetty, or Undertow, eliminating the need for external server setup.
3. **Standalone Applications:** Allows running Spring applications with just a single command (`java -jar app.jar`).
4. **Production-Ready Features:** Includes monitoring, health checks, and metrics with **Spring Actuator**.
5. **Microservices-Friendly:** Easily supports **Microservices Architecture** with minimal setup.

6. **Spring Ecosystem Integration:** Works well with Spring components like Spring Data, Spring Security, and Spring Cloud.
7. **Faster Development:** Reduces boilerplate code and increases developer productivity.
8. **Externalized Configuration:** Supports `application.properties` / `YAML` for environment-specific settings.
9. **Built-in Dependency Management:** Uses `Spring Boot Starter` dependencies to simplify adding required libraries.
10. **Reduces XML Configuration:** Uses Java-based annotations (`@SpringBootApplication`, `@RestController`, `@Service`, `@Repository`) instead of XML.

Benefits of Spring Boot for Application Development

1 Rapid Development

- Auto-configures most dependencies, reducing manual setup.
- Comes with a built-in application server, so no need to deploy externally.
- Supports `Spring Initializr` to create projects quickly.

2 Microservices Ready

- Ideal for **microservices** architecture.
- Supports `Spring Cloud` for distributed applications.
- Lightweight and scalable.

3 Embedded Servers

- No need for external Tomcat or Jetty installation.
- Comes with **embedded Tomcat, Jetty, or Undertow**.



4 Production-Ready Features

- Spring Boot Actuator provides built-in **health checks, metrics, and monitoring**.
- Logging and security features are pre-configured.

5 Easy Integration

- Works seamlessly with **Spring MVC, Spring Data, JPA, Hibernate, REST APIs**.
- Supports **Thymeleaf, JSP**, and frontend frameworks like React/Angular.

6 Less XML Configuration

- Uses **Java-based configuration** instead of complex `applicationContext.xml`.
- Reduces boilerplate code with **annotations** (`@SpringBootApplication`, `@RestController`).

7 Spring Boot CLI & DevTools

- **Spring Boot CLI** lets you run applications quickly using Groovy scripts.
- **Spring Boot DevTools** enables **hot-reloading** for fast development.



Use of Spring Boot

- Web Application Development (REST APIs, MVC-based applications).
- Microservices Architecture.
- Cloud-Based Applications (integrates with Docker, Kubernetes, AWS, etc.).
- Enterprise Applications (Banking, E-commerce, etc.).
- Data Processing Applications using Spring Batch.
- Real-time systems like chat applications.

What Happens If We Don't Use Spring Boot?

- **More Manual Configuration:** You must manually configure databases, servers, and security settings.
- **Slower Development:** Requires more code and XML configurations.
- **Complex Dependency Management:** You have to manage libraries and their versions manually.
- **Difficult Deployment:** Need external servers like Tomcat or WildFly for running applications.
- **Scalability Issues:** Microservices development becomes more complex without Spring Boot's built-in features.

- **Harder Integration:** Connecting with databases, security, messaging queues (Kafka, RabbitMQ) becomes more difficult.

Spring Boot vs Traditional Spring

Feature	Spring Framework	Spring Boot
Configuration	Manual XML/Java	Auto-configured
Server	External Tomcat/Jetty	Embedded Tomcat/Jetty
Microservices Support	Needs setup	Built-in support
REST API Development	Requires setup	Easy with annotations
Performance	More manual effort	Optimized for speed

Conclusion

Spring Boot simplifies enterprise application development by providing **ready-to-use configurations, embedded servers, and production-ready features**. It speeds up development, reduces complexity, and is widely used for **Microservices and Cloud-based applications**

spring boot

framework built on top of spring f/w
 ->simplifies the development of java based applications
 by reducing boilerplate code and configuration.

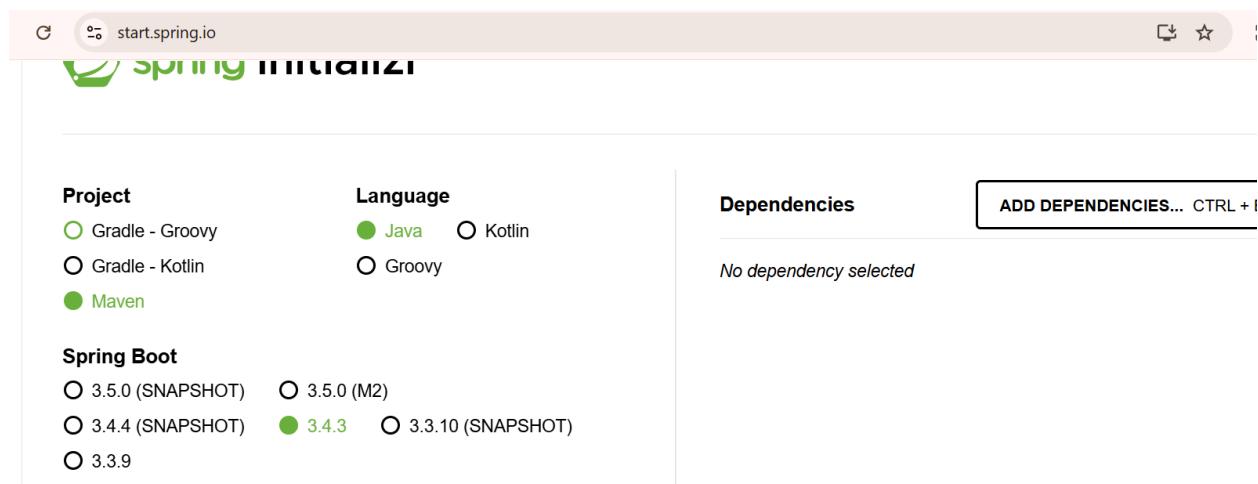
+

mainly used for building standalone
 microservices-based applications
 production-ready

key features of springboot:

1. auto configuration
2. microservices support
3. starter dependencies
4. embeded servers
5. spring boot cli
6. production ready features
7. standalone application

To develop a spring boot application we r having one website that is start.spring.io



The screenshot shows the configuration interface for a Spring Boot application. The top navigation bar includes a back button, a refresh button, and a star icon. The main title is "Spring Initializr".

Project (radio buttons):
 Gradle - Groovy
 Gradle - Kotlin
 Maven

Language (radio buttons):
 Java
 Kotlin
 Groovy

Dependencies (button): ADD DEPENDENCIES... CTRL + I
No dependency selected

Spring Boot (radio buttons):
 3.5.0 (SNAPSHOT)
 3.4.4 (SNAPSHOT)
 3.4.3
 3.3.9
 3.5.0 (M2)
 3.3.10 (SNAPSHOT)

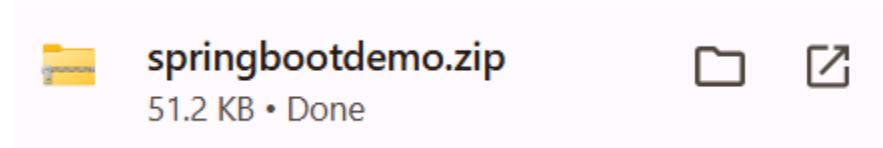
Project Metadata

Group	com.wipro
Artifact	springbootdemo
Name	springbootdemo
Description	Demo project for Spring Boot
Package name	com.wipro
Packaging	<input checked="" type="radio"/> Jar <input type="radio"/> War
Java	<input type="radio"/> 23 <input type="radio"/> 21 <input checked="" type="radio"/> 17

Then click on the below button

GENERATE CTRL + ↵

Once we click on the generate then it will download the zip file

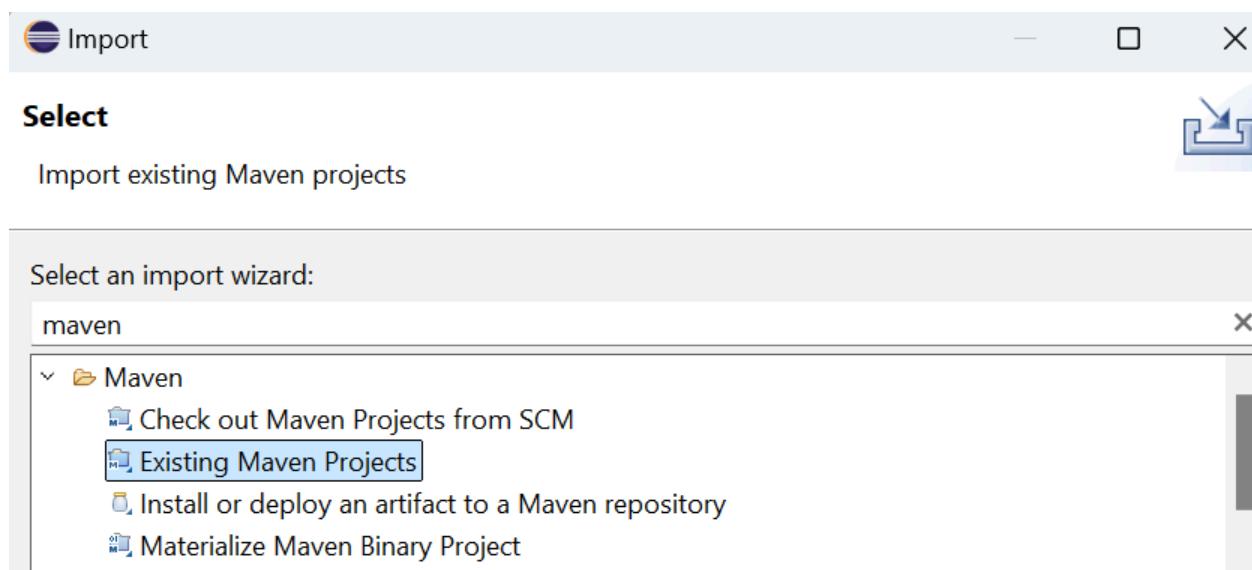


Now extract the zip file in the eclipse files folder

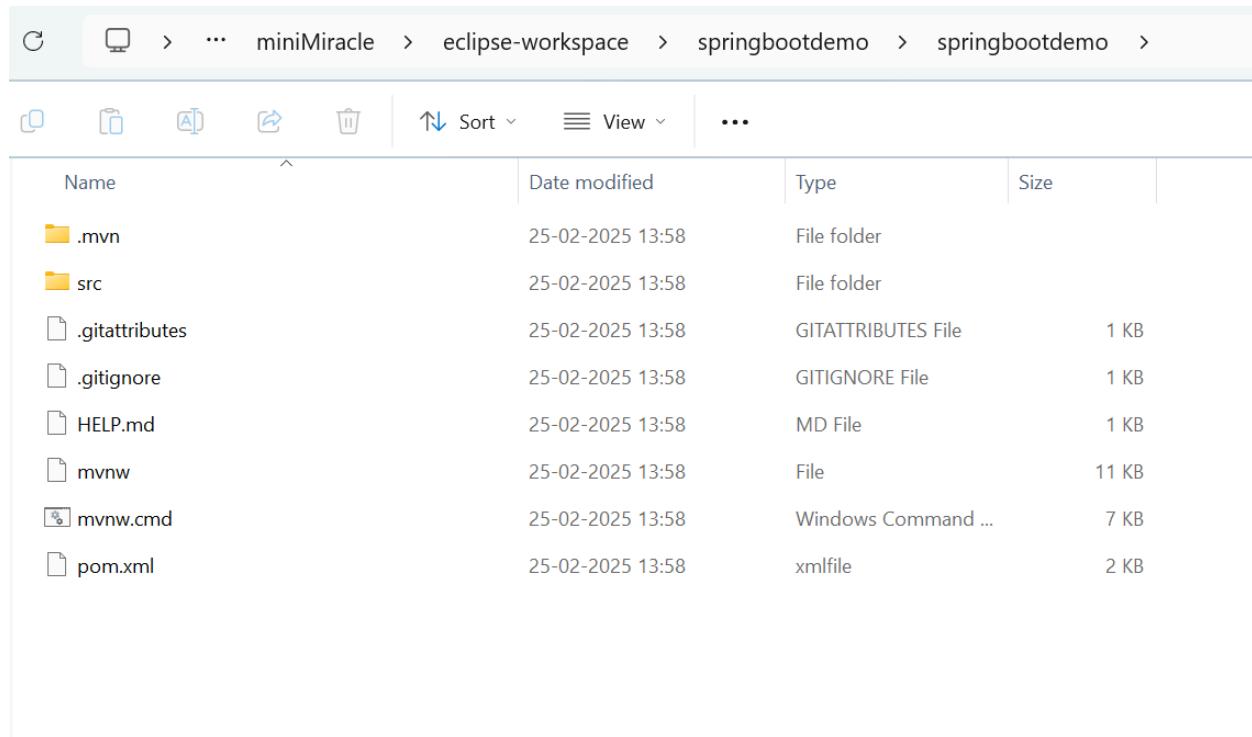
Name	Date modified	Type	Size
gradle	25-02-2025 13:48	File folder	
src	25-02-2025 13:48	File folder	
.gitattributes	25-02-2025 13:48	GITATTRIBUTES File	1 KB
.gitignore	25-02-2025 13:48	GITIGNORE File	1 KB
build.gradle	25-02-2025 13:48	GRADLE File	1 KB
gradlew	25-02-2025 13:48	File	9 KB
gradlew.bat	25-02-2025 13:48	Windows Batch File	3 KB
HELP.md	25-02-2025 13:48	MD File	1 KB
settings.gradle	25-02-2025 13:48	GRADLE File	1 KB

This is the folder structure once we extracted the zip file

Now we need to import into the eclipse



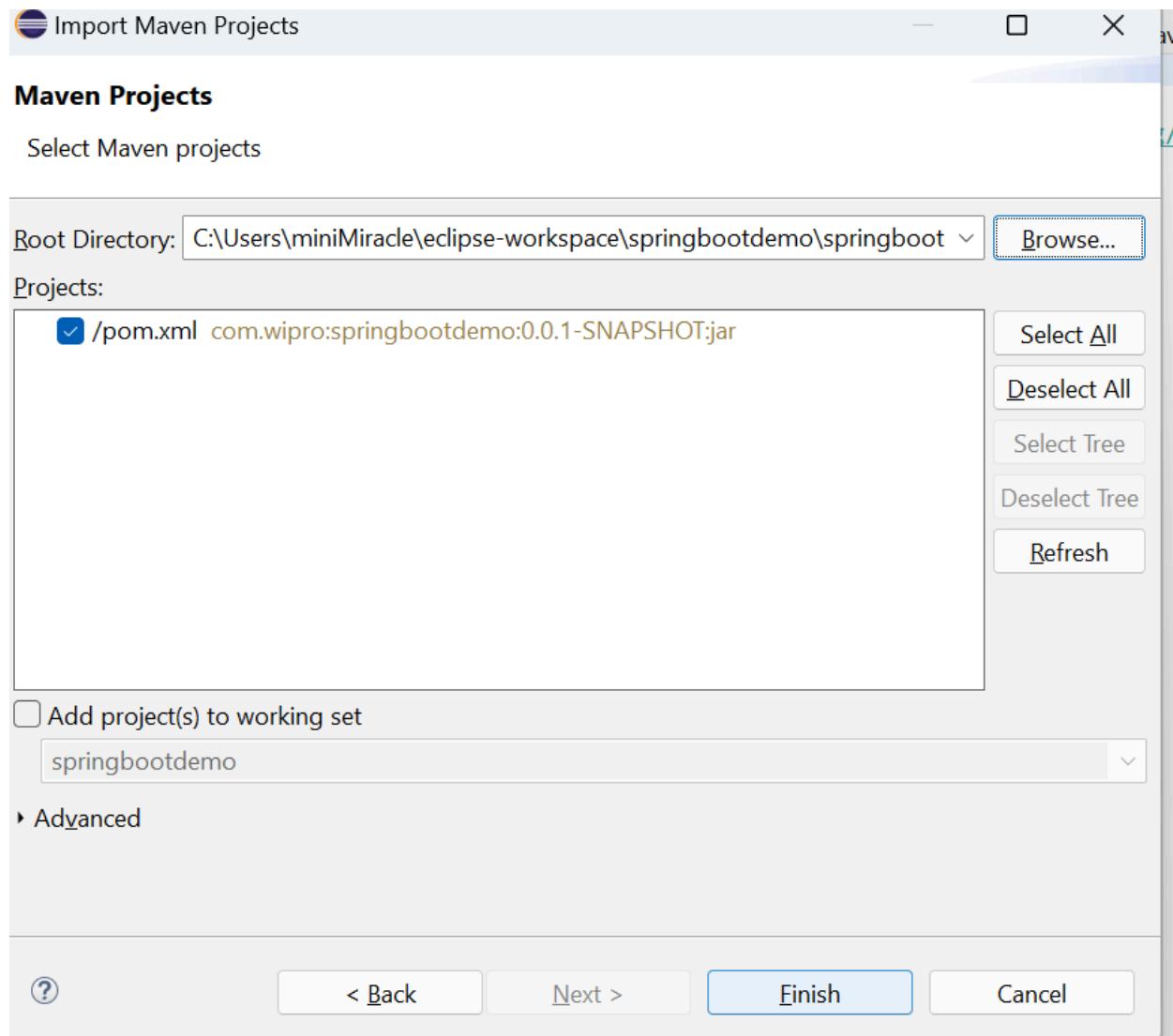
now click on the select folder



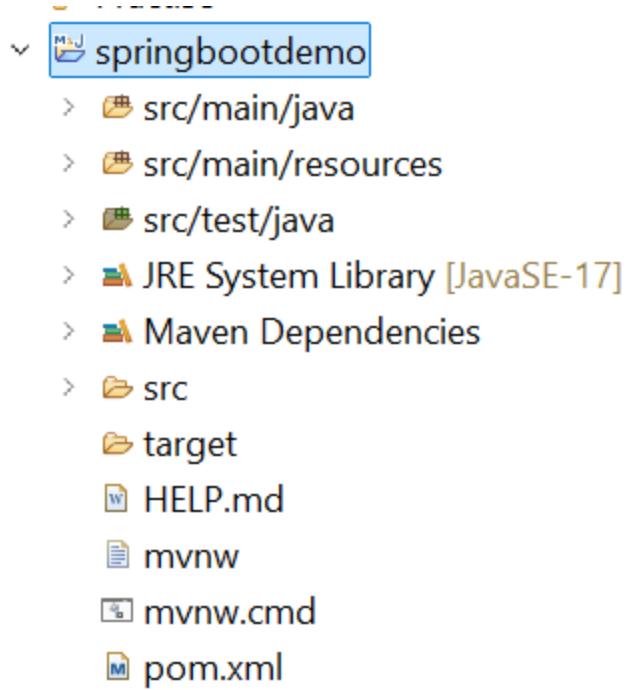
The screenshot shows a file explorer window with the following directory structure and file list:

```
miniMiracle > eclipse-workspace > springbootdemo > springbootdemo
```

Name	Date modified	Type	Size
.mvn	25-02-2025 13:58	File folder	
src	25-02-2025 13:58	File folder	
.gitattributes	25-02-2025 13:58	GITATTRIBUTES File	1 KB
.gitignore	25-02-2025 13:58	GITIGNORE File	1 KB
HELP.md	25-02-2025 13:58	MD File	1 KB
mvnw	25-02-2025 13:58	File	11 KB
mvnw.cmd	25-02-2025 13:58	Windows Command ...	7 KB
pom.xml	25-02-2025 13:58	xmlfile	2 KB



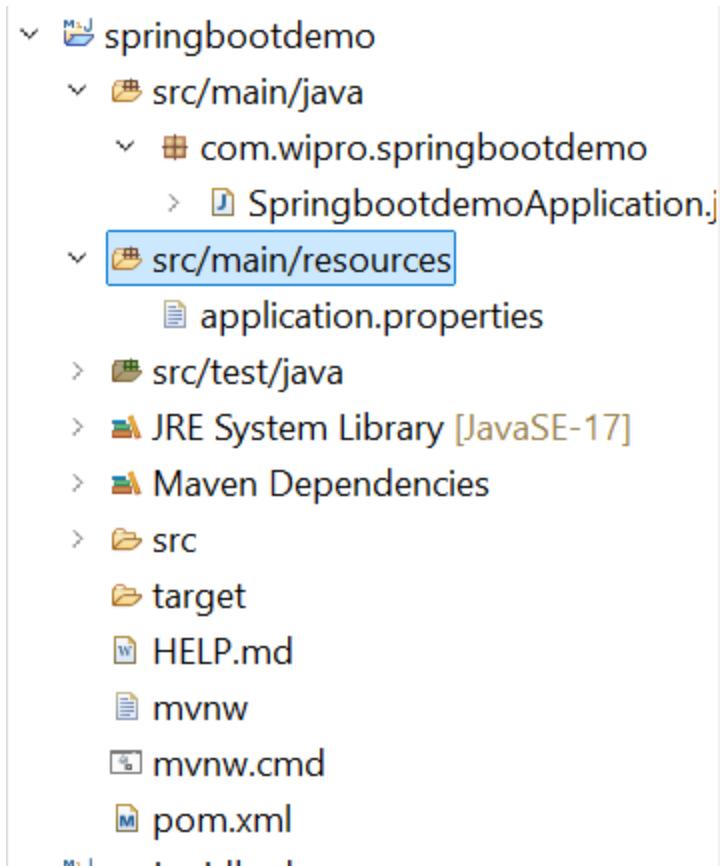
then automatically pom.xml file is generated..then click on the finish button



Here all the files are created

- >  [accessors-smart-2.5.2.jar](#) - C:\Users\miniMira\IdeaProjects\Kotlin\app\libs\accessors-smart-2.5.2.jar
- >  [android-json-0.0.20131108.vaadin.jar](#) - C:\Users\miniMira\IdeaProjects\Kotlin\app\libs\android-json-0.0.20131108.vaadin.jar
- >  [apiguardian-api-1.1.2.jar](#) - C:\Users\miniMira\IdeaProjects\Kotlin\app\libs\apiguardian-api-1.1.2.jar
- >  [asm-9.7.1.jar](#) - C:\Users\miniMira\IdeaProjects\Kotlin\app\libs\asm-9.7.1.jar
- >  [assertj-core-3.26.3.jar](#) - C:\Users\miniMira\IdeaProjects\Kotlin\app\libs\assertj-core-3.26.3.jar
- >  [Awaitility-4.2.2.jar](#) - C:\Users\miniMira\IdeaProjects\Kotlin\app\libs\Awaitility-4.2.2.jar
- >  [byte-buddy-1.15.11.jar](#) - C:\Users\miniMira\IdeaProjects\Kotlin\app\libs\byte-buddy-1.15.11.jar
- >  [byte-buddy-agent-1.15.11.jar](#) - C:\Users\miniMira\IdeaProjects\Kotlin\app\libs\byte-buddy-agent-1.15.11.jar
- >  [hamcrest-2.2.jar](#) - C:\Users\miniMira\IdeaProjects\Kotlin\app\libs\hamcrest-2.2.jar
- >  [jakarta.activation-api-2.1.3.jar](#) - C:\Users\miniMira\IdeaProjects\Kotlin\app\libs\jakarta.activation-api-2.1.3.jar
- >  [jakarta.annotation-api-2.1.1.jar](#) - C:\Users\miniMira\IdeaProjects\Kotlin\app\libs\jakarta.annotation-api-2.1.1.jar
- >  [jakarta.xml.bind-api-4.0.2.jar](#) - C:\Users\miniMira\IdeaProjects\Kotlin\app\libs\jakarta.xml.bind-api-4.0.2.jar
- >  [jsonassert-1.5.3.jar](#) - C:\Users\miniMira\IdeaProjects\Kotlin\app\libs\jsonassert-1.5.3.jar
- >  [json-path-2.9.0.jar](#) - C:\Users\miniMira\IdeaProjects\Kotlin\app\libs\json-path-2.9.0.jar
- >  [json-smart-2.5.2.jar](#) - C:\Users\miniMira\IdeaProjects\Kotlin\app\libs\json-smart-2.5.2.jar
- >  [jul-to-slf4j-2.0.16.jar](#) - C:\Users\miniMira\IdeaProjects\Kotlin\app\libs\jul-to-slf4j-2.0.16.jar
- >  [junit-jupiter-5.11.4.jar](#) - C:\Users\miniMira\IdeaProjects\Kotlin\app\libs\junit-jupiter-5.11.4.jar
- >  [junit-jupiter-api-5.11.4.jar](#) - C:\Users\miniMira\IdeaProjects\Kotlin\app\libs\junit-jupiter-api-5.11.4.jar
- >  [junit-jupiter-engine-5.11.4.jar](#) - C:\Users\miniMira\IdeaProjects\Kotlin\app\libs\junit-jupiter-engine-5.11.4.jar
- >  [junit-jupiter-params-5.11.4.jar](#) - C:\Users\miniMira\IdeaProjects\Kotlin\app\libs\junit-jupiter-params-5.11.4.jar
- >  [junit-platform-commons-1.11.4.jar](#) - C:\Users\miniMira\IdeaProjects\Kotlin\app\libs\junit-platform-commons-1.11.4.jar
- >  [junit-platform-engine-1.11.4.jar](#) - C:\Users\miniMira\IdeaProjects\Kotlin\app\libs\junit-platform-engine-1.11.4.jar
- >  [log4j-api-2.24.3.jar](#) - C:\Users\miniMira\IdeaProjects\Kotlin\app\libs\log4j-api-2.24.3.jar
- >  [log4j-to-slf4j-2.24.3.jar](#) - C:\Users\miniMira\IdeaProjects\Kotlin\app\libs\log4j-to-slf4j-2.24.3.jar
- >  [logback-classic-1.5.16.jar](#) - C:\Users\miniMira\IdeaProjects\Kotlin\app\libs\logback-classic-1.5.16.jar
- >  [logback-core-1.5.16.jar](#) - C:\Users\miniMira\IdeaProjects\Kotlin\app\libs\logback-core-1.5.16.jar
- >  [micrometer-commons-1.14.4.jar](#) - C:\Users\miniMira\IdeaProjects\Kotlin\app\libs\micrometer-commons-1.14.4.jar

All the dependencies jar files will be pulled automatically by default

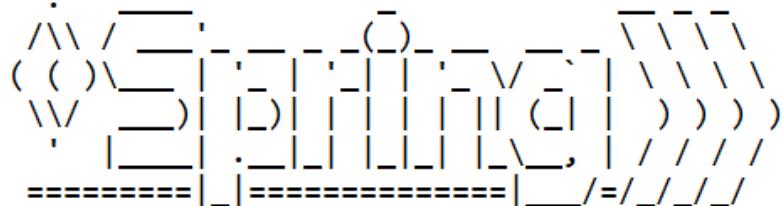


Now go to the SpringbootApplication and run as java application

```
SpringbootdemoApplication.java
1 package com.wipro.springbootdemo;
2
3 import org.springframework.boot.SpringApplication;
4
5 @SpringBootApplication
6 public class SpringbootdemoApplication {
7
8     public static void main(String[] args) {
9         SpringApplication.run(SpringbootdemoApplication.class, args);
10    }
11
12
13 }
14
```

Now my requirement is i want to see the hello world in the console

```
SpringbootdemoApplication.java ×
1 package com.wipro.springbootdemo;
2
3 import org.springframework.boot.SpringApplication;
4
5
6 @SpringBootApplication
7 public class SpringbootdemoApplication {
8
9     public static void main(String[] args) {
10         SpringApplication.run(SpringbootdemoApplication.class, args);
11         System.out.println("welcome to spring boot");
12     }
13
14 }
```



:: Spring Boot ::

(v3.4.3)

```
2025-02-25T14:10:58.525+05:30  INFO 13984 --- [springbootdemo] [  
2025-02-25T14:10:58.529+05:30  INFO 13984 --- [springbootdemo] [  
2025-02-25T14:10:59.237+05:30  INFO 13984 --- [springbootdemo] [  
welcome to spring boot
```

Execution flow of the Spring boot

1. Load and Initialize the JVM

- The Java Virtual Machine (JVM) loads the application class.
- `main()` method execution begins.

Why the control comes to the main method only...??

Because this class is annotated with the Springboot application

```
SpringbootdemoApplication.java ×
1 package com.wipro.springbootdemo;
2
3 import org.springframework.boot.SpringApplication;
4
5 @SpringBootApplication
6 public class SpringbootdemoApplication {
7
8     public static void main(String[] args) {
9         SpringApplication.run(SpringbootdemoApplication.class, args);
10        System.out.println("welcome to spring boot");
11    }
12
13
14 }
15
```

2. Spring Boot Application Startup (@SpringBootApplication)

- The `@SpringBootApplication` annotation enables:
 - `@Configuration` : Marks the class as a source of bean definitions.
 - `@EnableAutoConfiguration` : Automatically configures beans based on dependencies.
 - `@ComponentScan` : Scans for Spring components in the package and subpackages.

1 Annotation ie **SpringBootApplication** is equal to the 3 Annotations...So we no need to mention the configuration,enableautoConfiguration and componentScan ..we simply mention the **SpringBootApplication** instead of those 3

Configuration ; it is used to the @bean classes

```
@SpringBootApplication
P @ org.springframework.boot.autoconfigure.SpringBootApplication

@Target(value={TYPE})
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters={@Filter(type=CUSTOM, classes={TypeExcludeFilter.class}), @Filter(type=CUSTOM,
} classes={AutoConfigurationExcludeFilter.class)}))
@Retention(value=RUNTIME)
@Documented
@Inherited

Indicates a configuration class that declares one or more @Bean methods and also triggers auto-configuration and
component scanning. This is a convenience annotation that is equivalent to declaring @SpringBootApplication.
```

Press 'F2' for focus

3. `SpringApplication.run()` Execution

”

- `SpringApplication.run()` initializes the application `context`.
- It performs the following:
 - Creates an instance of `SpringApplication`
 - Determines if the application is a web or not. ↴ `web` application.
- Loads application properties (`application.properties` or `application.yml`).
- Starts the embedded web server (Tomcat, Jetty, or Undertow).
 ↳
- Initializes the Spring context (`ApplicationContext`).

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

org.springframework.boot.SpringApplication

Class that can be used to bootstrap and launch a Spring application from a Java main method. By default class will perform the following steps to bootstrap your application:

- Create an appropriate `ApplicationContext` instance (depending on your classpath)
- Register a `CommandLinePropertySource` to expose command line arguments as Spring properties
- Refresh the application context, loading all singleton beans
- Trigger any `CommandLineRunner` beans

In most circumstances the static `run(Class, String [])` method can be called directly from your main method to bootstrap your application:

4. Spring Container Initialization

- The Spring container initializes beans and dependencies using:
 - **Bean creation:** Objects annotated with `@Component`, `@Service`, `@Repository`, etc., are created.
 - **Dependency Injection:** Beans are injected where required.
 - **Post-processing:** `@PostConstruct` and `InitializingBean` execute initialization logic.

5. Auto-Configuration (`@EnableAutoConfiguration`)

- Based on dependencies, Spring Boot auto-configures components such as:
 - **Spring MVC** (if `spring-web` dependency is present).
 - **Spring Data JPA** (if `spring-data-jpa` and a database driver are present).
 - **Spring Security** (if `spring-boot-starter-security` is included).

6. Embedded Web Server Startup (For Web Applications)



- If it's a web application, Spring Boot starts the embedded server:
 - Default: Tomcat (port 8080)
 - Others: Jetty, Undertow (can be configured)

7. DispatcherServlet Initialization (For Web Apps)

- The `DispatcherServlet` is initialized to handle incoming HTTP requests.
- It follows the `Front Controller` design pattern:
 1. Intercept request
 2. Find the appropriate controller
 3. Invoke business logic
 4. Return the response

8. Execution of Business Logic & Data Handling

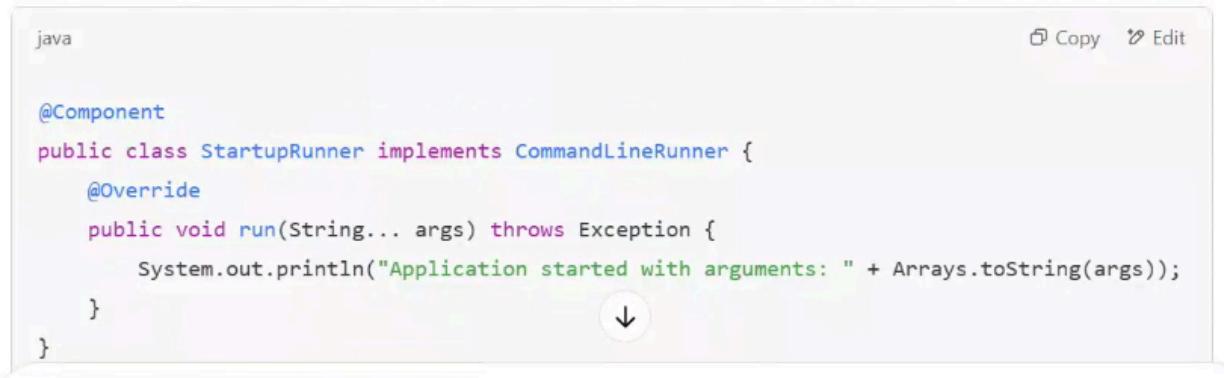
- Controllers (`@RestController` or `@Controller`) handle requests.
- Services (`@Service`) execute business logic.
- Repositories (`@Repository` with Spring Data JPA) perform database operations.

9. Response Generation & Serialization

- The response (JSON/XML/View) is generated and sent back to the client.
- If `@ResponseBody` or `@RestController` is used, JSON serialization happens via Jackson.

10. Running CommandLineRunner or ApplicationRunner (Optional)

- If implemented, these interfaces execute logic at startup.



```
java

@Component
public class StartupRunner implements CommandLineRunner {
    @Override
    public void run(String... args) throws Exception {
        System.out.println("Application started with arguments: " + Arrays.toString(args));
    }
}
```

The code editor shows a Java file with the following content:

```
java

@Component
public class StartupRunner implements CommandLineRunner {
    @Override
    public void run(String... args) throws Exception {
        System.out.println("Application started with arguments: " + Arrays.toString(args));
    }
}
```

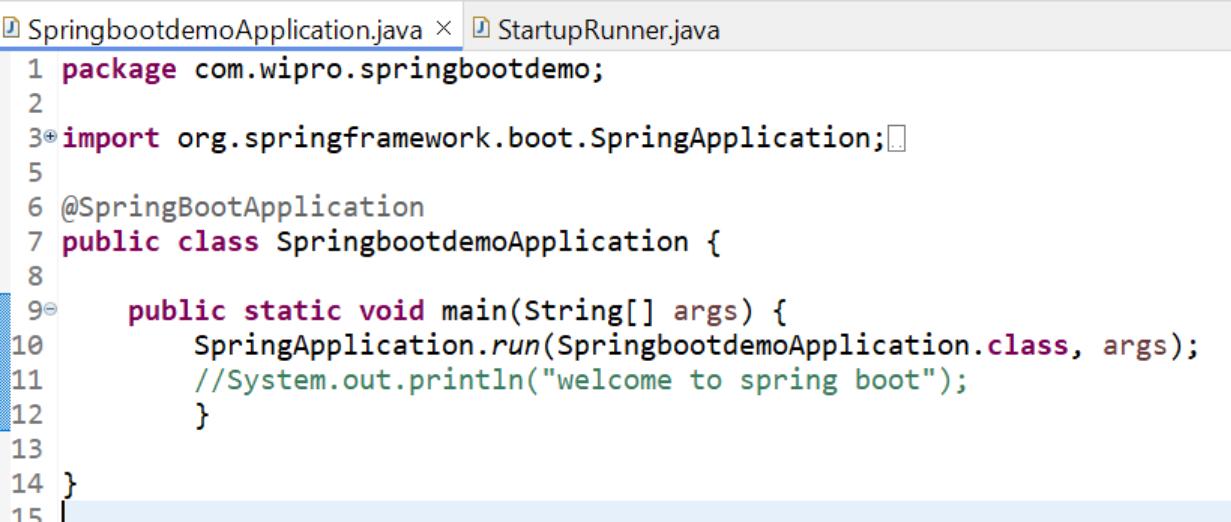
Buttons for 'Copy' and 'Edit' are visible in the top right corner.

what is commandlinerunner interface in springboot

CommandLineRunner is a functional interface in Spring Boot that allows you to execute code after the application context has been initialized. It is typically used for running logic at startup, such as seeding a database, setting up default configurations, or performing any initialization tasks.

Usage of CommandLineRunner

It has a single method.



```
SpringbootdemoApplication.java × StartupRunner.java
1 package com.wipro.springbootdemo;
2
3 import org.springframework.boot.SpringApplication;
4
5 @SpringBootApplication
6 public class SpringbootdemoApplication {
7
8     public static void main(String[] args) {
9         SpringApplication.run(SpringbootdemoApplication.class, args);
10        //System.out.println("welcome to spring boot");
11    }
12
13
14 }
15
```

The code editor shows two files: SpringbootdemoApplication.java and StartupRunner.java. The SpringbootdemoApplication.java file contains the main method:

```
SpringbootdemoApplication.java × StartupRunner.java
1 package com.wipro.springbootdemo;
2
3 import org.springframework.boot.SpringApplication;
4
5 @SpringBootApplication
6 public class SpringbootdemoApplication {
7
8     public static void main(String[] args) {
9         SpringApplication.run(SpringbootdemoApplication.class, args);
10        //System.out.println("welcome to spring boot");
11    }
12
13
14 }
15
```

```
1 package com.wipro.springbootdemo;
2
3 import org.springframework.boot.CommandLineRunner;
4 import org.springframework.stereotype.Component;
5 @Component
6 public class StartupRunner implements CommandLineRunner {
7
8     @Override
9     public void run(String... args) throws Exception {
10         System.out.println("Application started with CommandLineRunner");
11
12         for (String arg : args) {
13             System.out.println("Argument: " + arg);
14         }
15
16     }
17
18 }
19
```

Right click on the **SpringBootDemoApplication**>>runas>>java application

The image shows the Spring Boot logo, which is a grid of characters including slashes, parentheses, and underscores. Below the logo, the text ':: Spring Boot :: (v3.4.3)' is displayed. Underneath that, a timestamp and log message are shown: '2025-02-25T14:46:04.155+05:30 INFO 19920 --- [springbootdemo] [main] c.w.s.S' followed by two identical lines of the same log message.

Now try to run as configuration to the same file ie `springbootademoapplication` **and go to** the arguments and mention few arguments like `abc def ghi jkl` and click on apply and click on run

```
:: Spring Boot ::          (v3.4.3)

2025-02-25T14:49:58.519+05:30  INFO 22260 --- [springbootdemo] [           main] c.w.s.Spring
2025-02-25T14:49:58.521+05:30  INFO 22260 --- [springbootdemo] [           main] c.w.s.Spring
2025-02-25T14:49:59.426+05:30  INFO 22260 --- [springbootdemo] [           main] c.w.s.Spring
Application started with CommandLineRunner
Argument: abc
Argument: def
Argument: ghi
Argument: jkl
```

what is the difference between `@Controller` and `@RestController` annotations?

1. `@Controller`

- It is used to define a **Spring MVC controller** that handles web requests and returns a `view` (typically a JSP, Thymeleaf, or any other UI template).
- Methods in a class annotated with `@Controller` return `logical view names` instead of JSON or XML responses.
- If you need to return data (e.g., JSON or XML), you must use `@ResponseBody` explicitly on methods.

Example:

2. `@RestController`

- It is a specialized version of `@Controller` introduced in **Spring 4.0**.
- It is a combination of `@Controller` and `@ResponseBody`, meaning that all methods inside the class automatically return `JSON` or `XML` instead of a view.
- This annotation is primarily used for **RESTful APIs**.

Key Differences

Feature	@Controller	@RestController
Purpose	Handles web UI requests (JSP, Thymeleaf)	Exposes RESTful web services
View Resolution	Returns logical view names	Returns JSON/XML directly
@ResponseBody	Required to return JSON/XML	Implicitly applied to all methods
Use Case	Web applications (JSP, Thymeleaf, etc.)	REST APIs (JSON/XML responses)

what is webservice? why should i develop my service as webservice
if i develop like webservice what will be the benefits in layman
terms explain me?

What is a Web Service?

A **web service** is a way for two different **applications** to talk to each other over the internet using **standard format** like **JSON** or **XML**. Imagine it like a **waiter in a restaurant**—you (the client) place an order (request), and the waiter (web service) brings you the food (response) from the kitchen (server).

Placing the order is request

Getting the food is response

us=clients

waiter=we services

Why Should I Develop My Service as a Web Service?

If you develop your application as a **web service**, it means your service can be accessed **from anywhere** and **by any system**, not just from your own app.

Benefits of Web Services (Layman's Terms)

1. Platform Independence (Works Everywhere)

- A web **service** allows your system to be accessed by **any device or application**—whether it's a **mobile app, website, or another service**—without needing changes in your code.
- Example: A bank's system can be accessed via mobile apps, websites, and ATMs using the same web service.

2. Language Independence (Use Any Technology)

- You can write your service in **Java, Python, .NET, or any language**, and **any client** (Android, iOS, web app) can use it.
- Example: A stock market API written in Java can be used by an Android app written in Kotlin or an iPhone app written in Swift.

3. Reusability (One Code, Many Uses)

- Instead of writing separate logic for different platforms (mobile, web, desktop), you write **one web service** that serves all of them.
- Example: A shopping app uses the same web service for its **website, mobile app, and admin dashboard**.

6. Integration (Connect with Other Services)

- You can **connect with third-party services** like **Google Maps, PayPal, or weather APIs** easily.
- Example: A **food delivery app** can integrate with Google Maps API to show restaurant locations.

SOAP VS RESTFUL API

1 SOAP Web Services (Old & Formal)

- Full Form: Simple Object Access Protocol (SOAP)
 - Data Format: Uses XML for request and response.
 - Transport Protocols: Works over HTTP, HTTPS, SMTP, FTP, etc.
 - Security: Supports WS-Security (Good for financial & enterprise applications).
 - Standards: Uses WSDL (Web Services Description Language) to define the service.
-
- Example: Used in banking, telecom, insurance, and enterprise applications.
 - Implementation in Java: JAX-WS (Java API for Web Services)

2 RESTful Web Services (Modern & Lightweight)

- Full Form: Representational State Transfer (REST)
- Data Format: Uses JSON (preferred) or XML.
- Transport Protocols: Works over HTTP/HTTPS.
- Security: Uses JWT, OAuth, Keys, or Basic Authentication.
- Standards: No WSDL, just a URL-based API.
- Example: Used in web apps, mobile apps, microservices, social media APIs (Facebook, Twitter, Google, etc.).

Web Services RestFul API



Project Gradle - Groovy Gradle - Kotlin Maven **Language** Java Kotlin Groovy

Spring Boot 3.5.0 (SNAPSHOT) 3.5.0 (M2) 3.4.4 (SNAPSHOT) 3.4.3 3.3.10 (SNAPSHOT) 3.3.9

Project Metadata

Group

Artifact

Name

Description

Package name

Packaging Jar War

Java 23 21 17

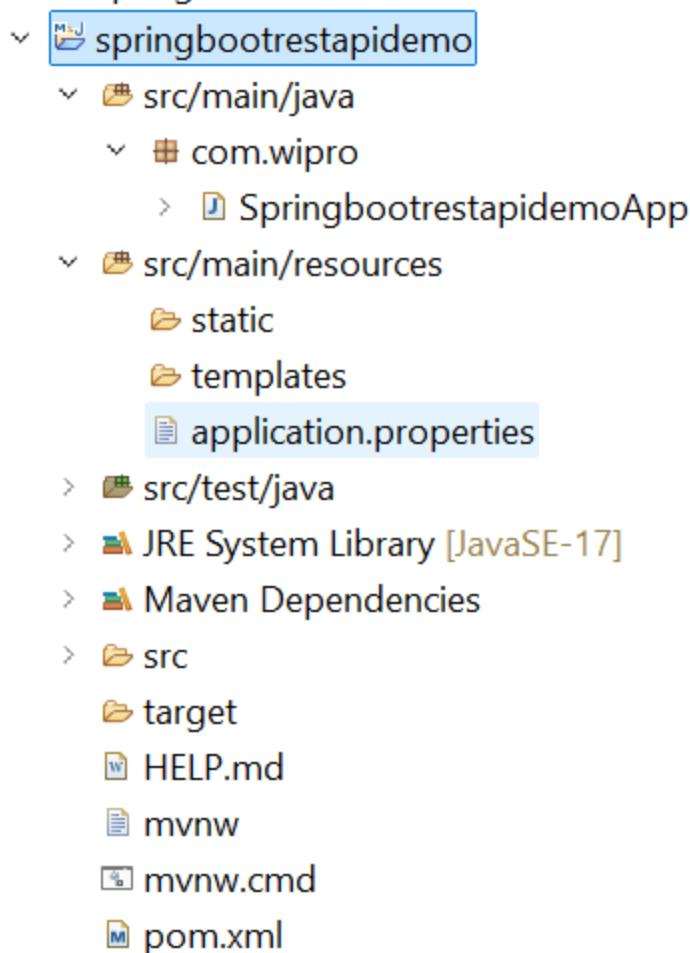
Dependencies**ADD DEPENDENCIES... CTRL + B****Spring Web** WEB

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

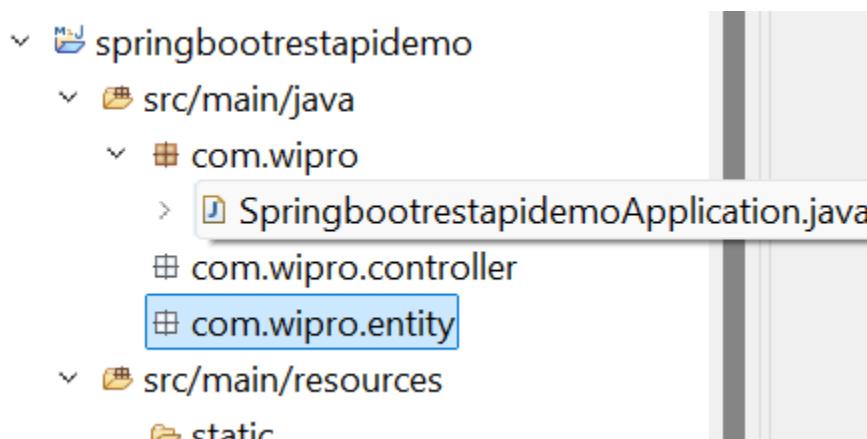
Spring Boot DevTools DEVELOPER TOOLS

Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

Once it is downloaded extract and import into the eclipse



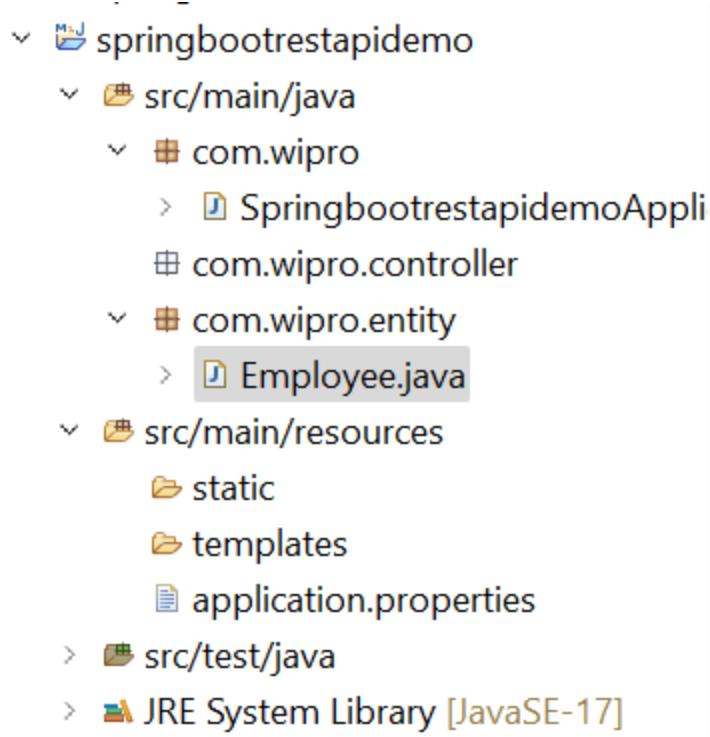
Now create two packages named **com.wipro.controller** and **com.wipro.entity**



Now create one employee class in entity package named as employee

Employee.java ×

```
1 package com.wipro.entity;
2 public class Employee {
3     private int id;
4     private String name;
5     private String department;
6
7     public Employee() {}
8
9     public Employee(int id, String name, String department) {
10         this.id = id;
11         this.name = name;
12         this.department = department;
13     }
14
15     public int getId() {
16         return id;
17     }
18
19     public void setId(int id) {
20         this.id = id;
21     }
22
23     public String getName() {
24         return name;
25     }
26
27     public void setName(String name) {
28         this.name = name;
29     }
30
31     public String getDepartment() {
32         return department;
33     }
34
35     public void setDepartment(String department) {
36         this.department = department;
37     }
38 }
39 }
```



Now we need to define the controller class in the controller package named as the EmployeeController

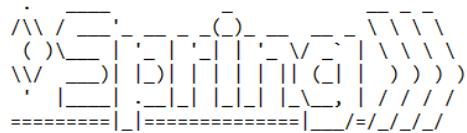
```
1 package com.wipro.controller;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 import org.springframework.web.bind.annotation.GetMapping;
7 import org.springframework.web.bind.annotation.PostMapping;
8 import org.springframework.web.bind.annotation.RequestBody;
9 import org.springframework.web.bind.annotation.RequestMapping;
10 import org.springframework.web.bind.annotation.RestController;
11
12 import com.wipro.entity.Employee;
13
14 @RestController
15 @RequestMapping("/employees") //this url is applicable to all the methods inside the class
16
17 public class EmployeeController
18 {
19
20     private List<Employee> employees = new ArrayList<>();
21     //prepopulate the data
22     public EmployeeController() {
23         employees.add(new Employee(100, "ram", "software"));
24         employees.add(new Employee(101, "Kumar", "HR"));
25     }
26
27     //fetch the data
28     @GetMapping
29     public List<Employee> getAllEmployees() {
30         return employees;
31     }
32     //if we want to add a new employee
33     @PostMapping
34     public String addEmployee(@RequestBody Employee employee) // as we sending employee object as http req
35                                         //into java object so we mentioned reqbody
36                                         //json to java object internally converted with RB
37     {
38         employees.add(employee); //once the employee obj is added then it adds into the list
39         return "Employee added successfully!"; //we gets this msg after adding
40     }
41     //if we pass localhost:9090/employees....which method will be exec 1.get or post....?
42     //to avoid those confusion we need to specify the req type like post or post type
43
44
45
46 }
47
```

```
1 package com.wipro;
2
3 import org.springframework.boot.SpringApplication;
4
5 @SpringBootApplication
6 public class SpringbootrestapidemoApplication {
7
8     public static void main(String[] args) {
9         SpringApplication.run(SpringbootrestapidemoApplication.class, args);
10    }
11
12 }
13
14
```

```
Employee.java EmployeeController.java application.p
1 spring.application.name=springbootrestapidemo
2 server.port=9090
3
```

springbootrestapidemo

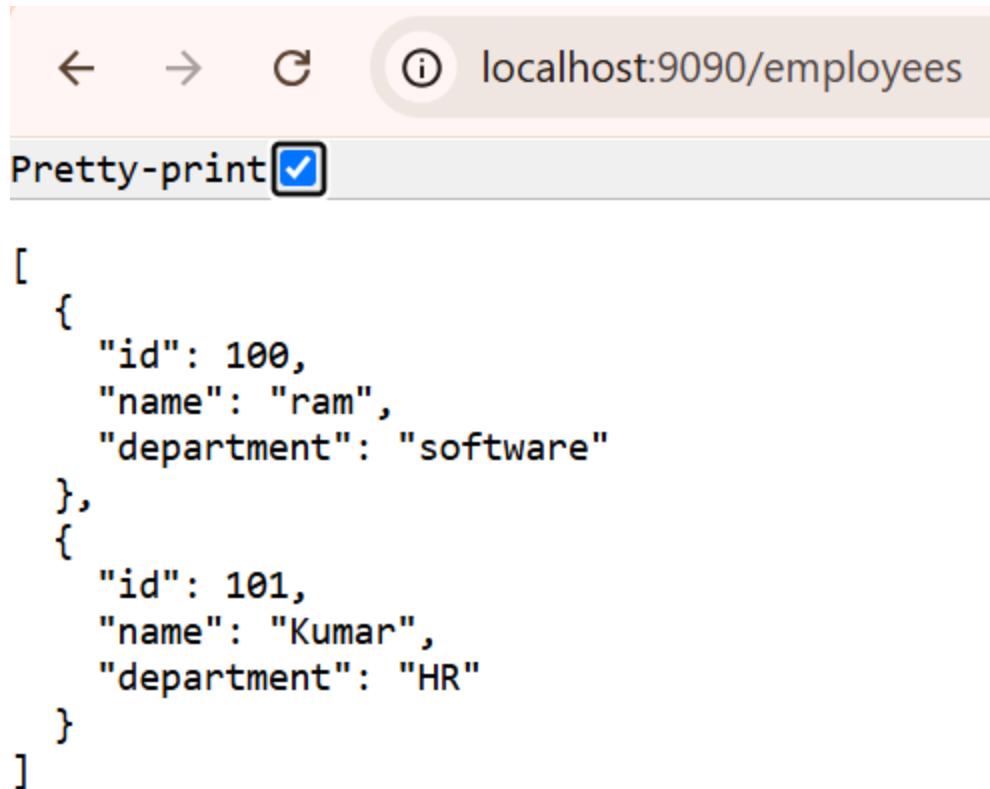
- src/main/java
 - com.wipro
 - SpringbootrestapidemoAppli
 - com.wipro.controller
 - EmployeeController.java
 - com.wipro.entity
 - Employee.java
- src/main/resources
 - static
 - templates
 - application.properties
- src/test/java
- JRE System Library [JavaSE-17]
- Maven Dependencies
- src
 - target
- HELP.md
- mvnw
- mvnw.cmd
- pom.xml



:: Spring Boot :: (v3.4.3)

```
025-02-25T16:58:41.758+05:30 INFO 18100 --- [springbootrestapidemo] [ restartedMain] c.w.Springbootrestapidemo
025-02-25T16:58:41.763+05:30 INFO 18100 --- [springbootrestapidemo] [ restartedMain] c.w.Springbootrestapidemo
025-02-25T16:58:41.926+05:30 INFO 18100 --- [springbootrestapidemo] [ restartedMain] e.DevToolsPropertyDefaul
025-02-25T16:58:41.926+05:30 INFO 18100 --- [springbootrestapidemo] [ restartedMain] e.DevToolsPropertyDefaul
025-02-25T16:58:44.079+05:30 INFO 18100 --- [springbootrestapidemo] [ restartedMain] o.s.b.w.embedded.tomcat.T
025-02-25T16:58:44.101+05:30 INFO 18100 --- [springbootrestapidemo] [ restartedMain] o.apache.catalina.core.St
025-02-25T16:58:44.102+05:30 INFO 18100 --- [springbootrestapidemo] [ restartedMain] o.apache.catalina.core.St
025-02-25T16:58:44.154+05:30 INFO 18100 --- [springbootrestapidemo] [ restartedMain] o.a.c.c.C.[Tomcat].[local
025-02-25T16:58:44.156+05:30 INFO 18100 --- [springbootrestapidemo] [ restartedMain] w.s.c.ServletWebServerApp
025-02-25T16:58:45.017+05:30 INFO 18100 --- [springbootrestapidemo] [ restartedMain] o.s.b.d.a.OptionalLiveRel
025-02-25T16:58:45.087+05:30 INFO 18100 --- [springbootrestapidemo] [ restartedMain] o.s.b.w.embedded.tomcat.T
025-02-25T16:58:45.103+05:30 INFO 18100 --- [springbootrestapidemo] [ restartedMain] c.w.Springbootrestapidemo
```

Server is started ..so now we need to check in browser by using **localhost:9090/employees**



A screenshot of a web browser window. The address bar shows 'localhost:9090/employees'. Below the address bar is a 'Pretty-print' checkbox which is checked. The main content area of the browser displays the following JSON data:

```
[  
  {  
    "id": 100,  
    "name": "ram",  
    "department": "software"  
  },  
  {  
    "id": 101,  
    "name": "Kumar",  
    "department": "HR"  
  }  
]
```

We passed the java object internally is is creating json and it is giving and vice versa ..It is only happening with the help of the **@RequestBody**

For suppose if we mention the id in the localhost like

localhost:9090/employees/101 then i need to get only that particular employee id's data only....How is it possible

It is possible with the help of the **path variables**

@PathVariable Annotation in Spring Boot

The `@PathVariable` annotation in Spring Boot is used to extract values from the URL path and map them to method parameters in a REST controller.



```
// GET API - Fetch Employee by ID
@GetMapping("/{id}")
public Employee getEmployeeById(@PathVariable int id){
    return employees.stream()
        .filter(emp -> emp.getId() == id)
        .findFirst()
        .orElse(null);
}

// POST API - Add a new employee
```

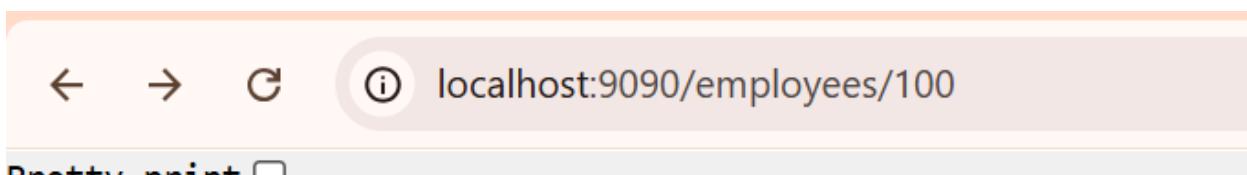
Whatever the id we r passing, if it matches with the employee id in the employee object ,that only we r fetching

```
@GetMapping("/{id}") //it is used to fetch the particular empl by using the id
public Employee getEmployeeById(@PathVariable int id)
{
    return employees.stream()
        .filter(emp -> emp.getId() == id)
        .findFirst()
        .orElse(null);
}
```

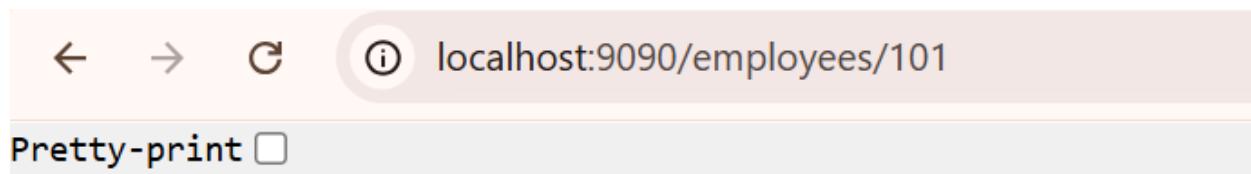
```

Employee.java EmployeeController.java application.properties Springbootrestapidemo
1 package com.wipro.controller;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 import org.springframework.web.bind.annotation.GetMapping;
7 import org.springframework.web.bind.annotation.PathVariable;
8 import org.springframework.web.bind.annotation.PostMapping;
9 import org.springframework.web.bind.annotation.RequestBody;
10 import org.springframework.web.bind.annotation.RequestMapping;
11 import org.springframework.web.bind.annotation.RestController;
12
13 import com.wipro.entity.Employee;
14
15 @RestController
16 @RequestMapping("/employees") //this url is applicable to all the methods inside the class
17
18 public class EmployeeController
19 {
20
21     private List<Employee> employees = new ArrayList<>();
22     //prepopulate the data
23     public EmployeeController()
24     {
25         employees.add(new Employee(100, "ram", "software"));
26         employees.add(new Employee(101, "Kumar", "HR"));
27     }
28
29     //fetch the data
30     @GetMapping
31     public List<Employee> getAllEmployees()
32     {
33         return employees;
34     }
35     //if we want to add a new employee
36     @PostMapping
37     public String addEmployee(@RequestBody Employee employee) // as we sending employee object as http req
38                                         //into java object so we mentioned reqbody
39                                         //json to java object internally converted with RB
40     {
41         employees.add(employee); //once the employee obj is added then it adds into the list
42         return "Employee added successfully!"; //we gets this msg after adding
43     }
44     //if we pass localhost:9090/employees....which method will be exec 1.get or post....?
45     //to avoid those confusion we need to specify the req type like post or post type
46     // GET API - Fetch employee by ID
47
48     @GetMapping("/{id}") //it is used to fetch the particular emp by using the id
49     public Employee getEmployeeById(@PathVariable int id)
50     {
51         return employees.stream()
52             .filter(emp -> emp.getId() == id)
53             .findFirst()
54             .orElse(null);
55     }
56
57
58
59 }

```



```
{"id":100, "name": "ram", "department": "software"}
```



A screenshot of a web browser window. The address bar shows 'localhost:9090/employees/101'. Below the address bar is a 'Pretty-print' checkbox. The main content area displays a JSON response: {"id":101, "name": "Kumar", "department": "HR"}

For Suppose if want to add the data then we can use the tools like POSTMAN

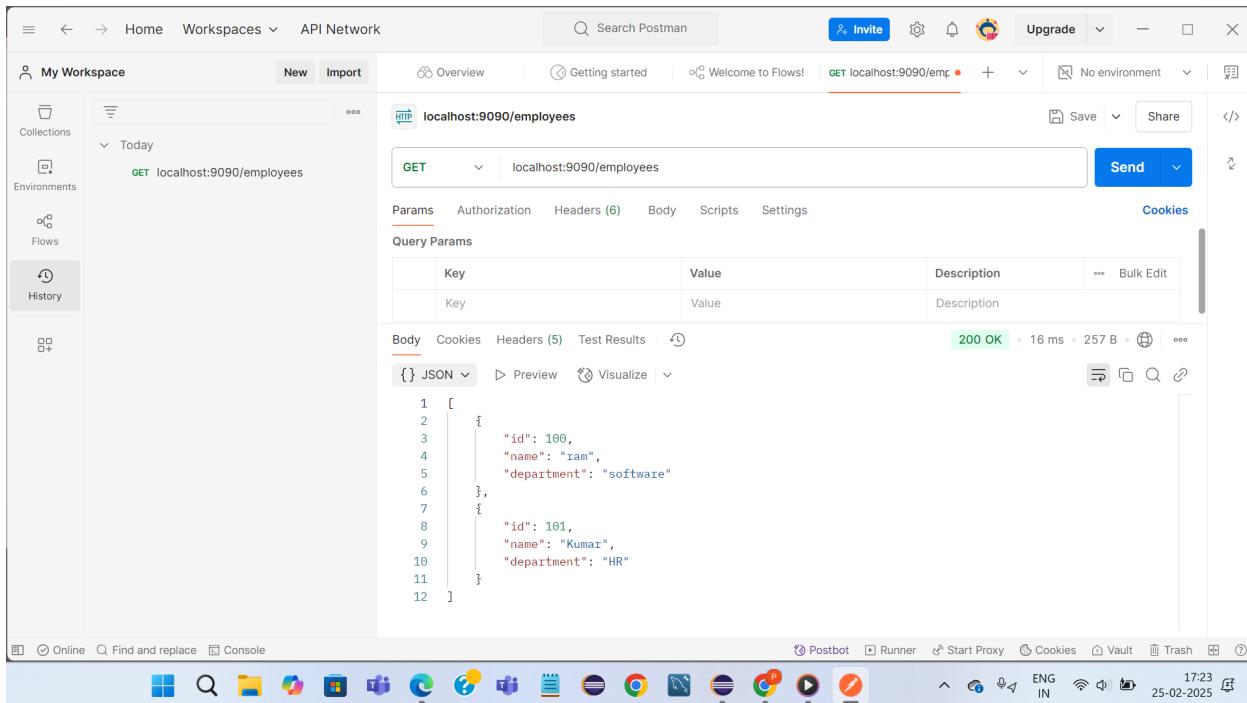
can you please list out some restful clients to test the rest url's

Sure! Here are some popular RESTful clients that you can use to test REST APIs:

1. Postman 🔥

- One of the most popular API testing tools.
- Supports GET, POST, PUT, DELETE, PATCH requests.
- Can handle authentication (JWT, OAuth, Basic Auth).
- Offers scripting with Pre-request & Test Scripts.
- Allows API automation and mock servers.

- Supports POST, PUT, DELETE, PATCH requests.
- Can handle authentication (JWT, OAuth, Basic Auth).
- Offers scripting with Pre-request & Test Scripts.
- Allows API automation and mock servers.
- Download: <https://www.postman.com/>



My Workspace

localhost:9090/employees

GET localhost:9090/employees

Params Authorization Headers (6) Body Scripts Settings

Query Params

Body Cookies Headers (5) Test Results

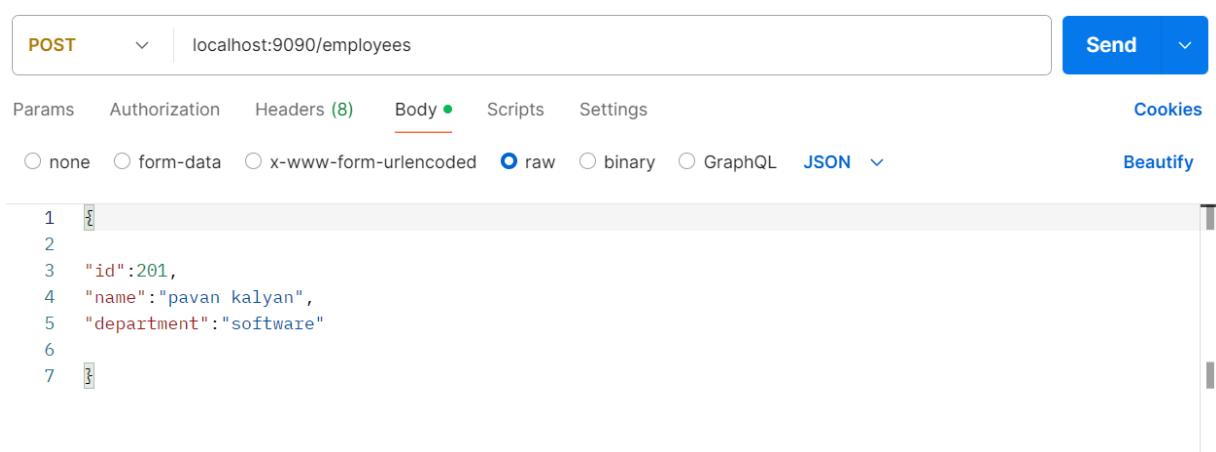
200 OK 16 ms 257 B

```
[{"id": 100, "name": "ram", "department": "software"}, {"id": 101, "name": "Kumar", "department": "HR"}]
```

Now i am getting the same output in the postman also.....

Now we need to add the data in the postman

Then choose the post method >> click on body>>click on raw



POST localhost:9090/employees

Params Authorization Headers (8) Body Scripts Settings

Body (raw)

```
{"id": 201, "name": "pavan kalyan", "department": "software"}
```

After clicking the send button

localhost:9090/employees

POST localhost:9090/employees Send

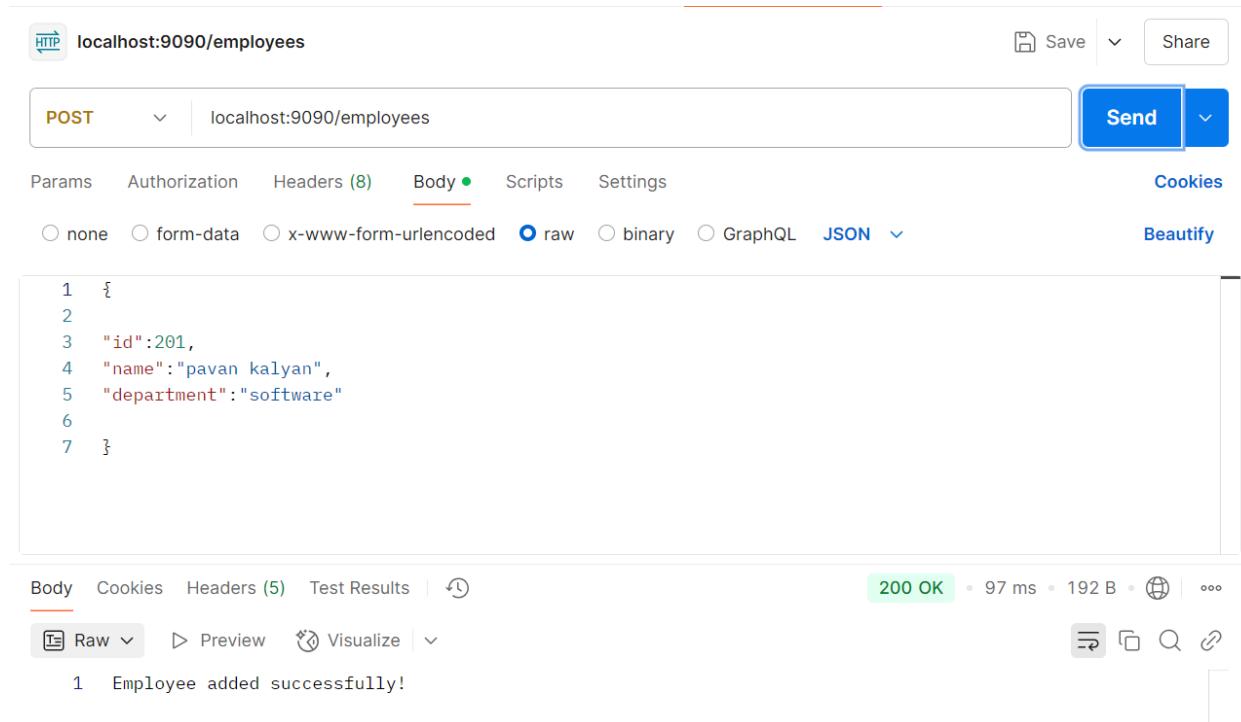
Params Authorization Headers (8) **Body** Scripts Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

```
1  {
2
3  "id":201,
4  "name":"pavan kalyan",
5  "department":"software"
6
7 }
```

Body Cookies Headers (5) Test Results 200 OK 97 ms 192 B

Raw Preview Visualize 1 Employee added successfully!



Now check whether the data is added or not by mentioning the method as **get**

GET localhost:9090/employees Send

Params Authorization Headers (8) **Body** Scripts Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

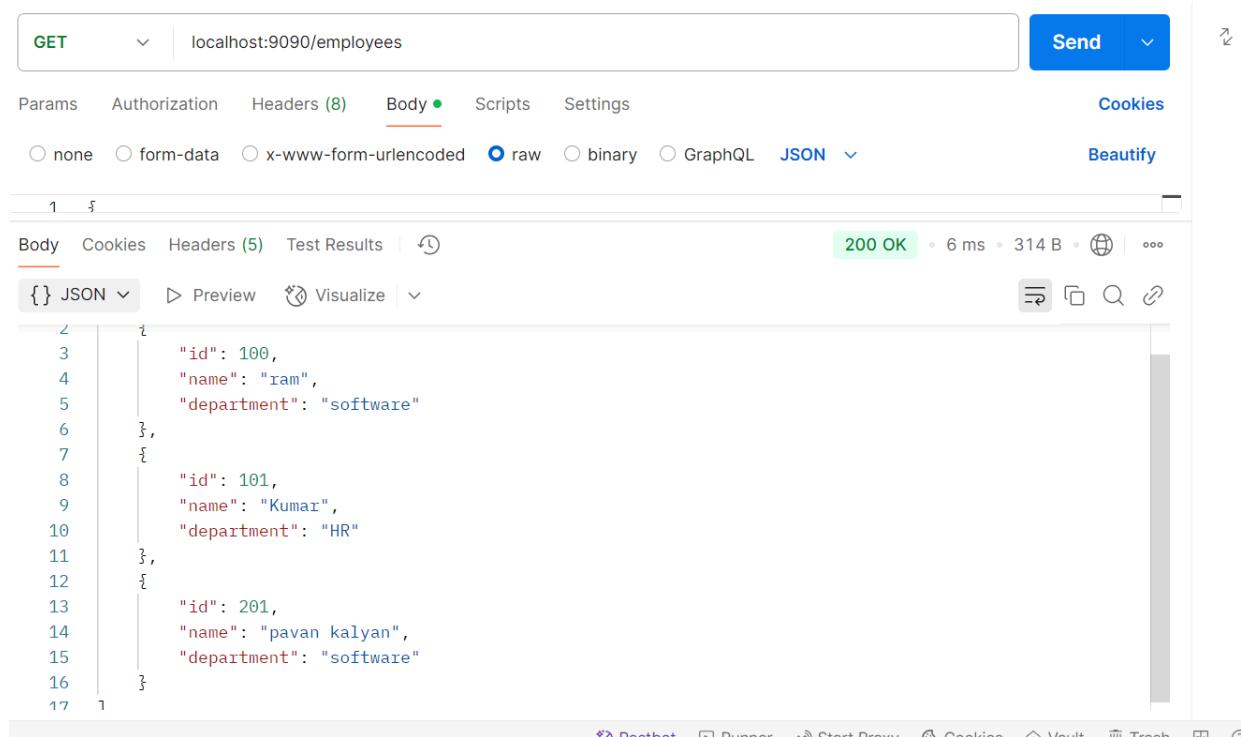
```
1 {
```

Body Cookies Headers (5) Test Results 200 OK 6 ms 314 B

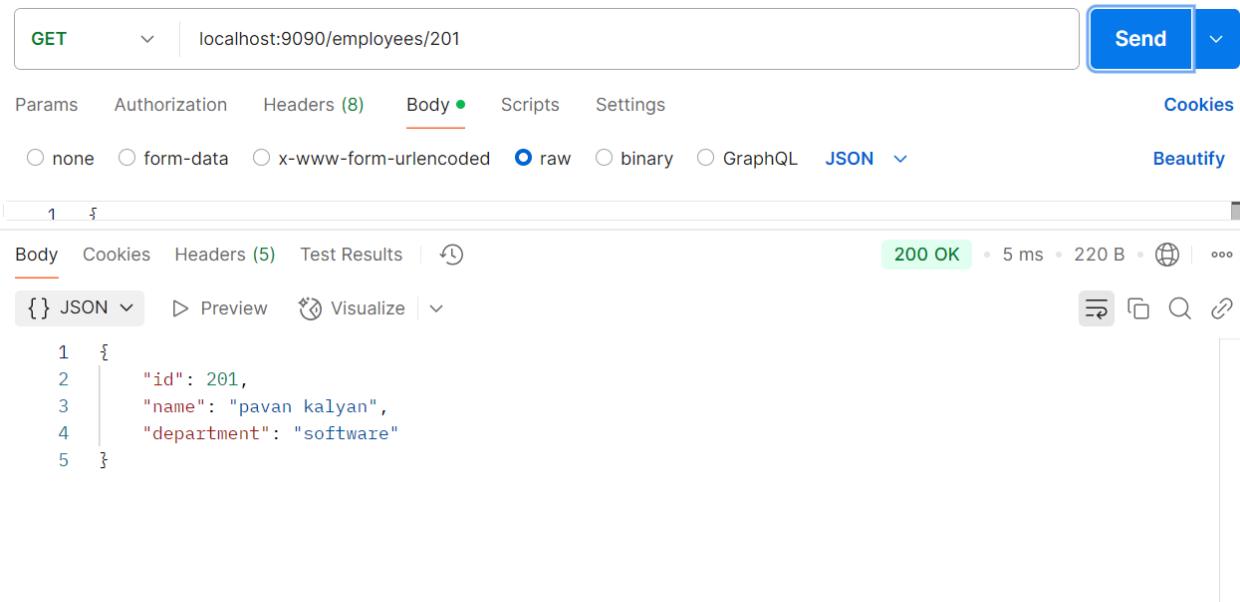
{ } JSON Preview Visualize

```
2 {
3   "id": 100,
4   "name": "ram",
5   "department": "software"
6 },
7 {
8   "id": 101,
9   "name": "Kumar",
10  "department": "HR"
11 },
12 {
13   "id": 201,
14   "name": "pavan kalyan",
15   "department": "software"
16 }
```

Postman Runner Start Drayv Cookies Vault Trash



Retrieving with the single id



The screenshot shows a REST client interface with the following details:

- Method:** GET
- URL:** localhost:9090/employees/201
- Headers:** (8)
- Body:** (selected)
- Params:** none
- Authorization:** none
- Headers (8):** none
- Body:** raw
- Scripts:** none
- Settings:** none
- Cookies:** none
- Beautify:** none

The response details are as follows:

- Status:** 200 OK
- Time:** 5 ms
- Size:** 220 B
- Content Type:** application/json
- More:** ...

The response body is displayed in JSON format:

```
1 {  
2   "id": 201,  
3   "name": "pavan kalyan",  
4   "department": "software"  
5 }
```