

Unit-5

Spring Framework

Spring Framework

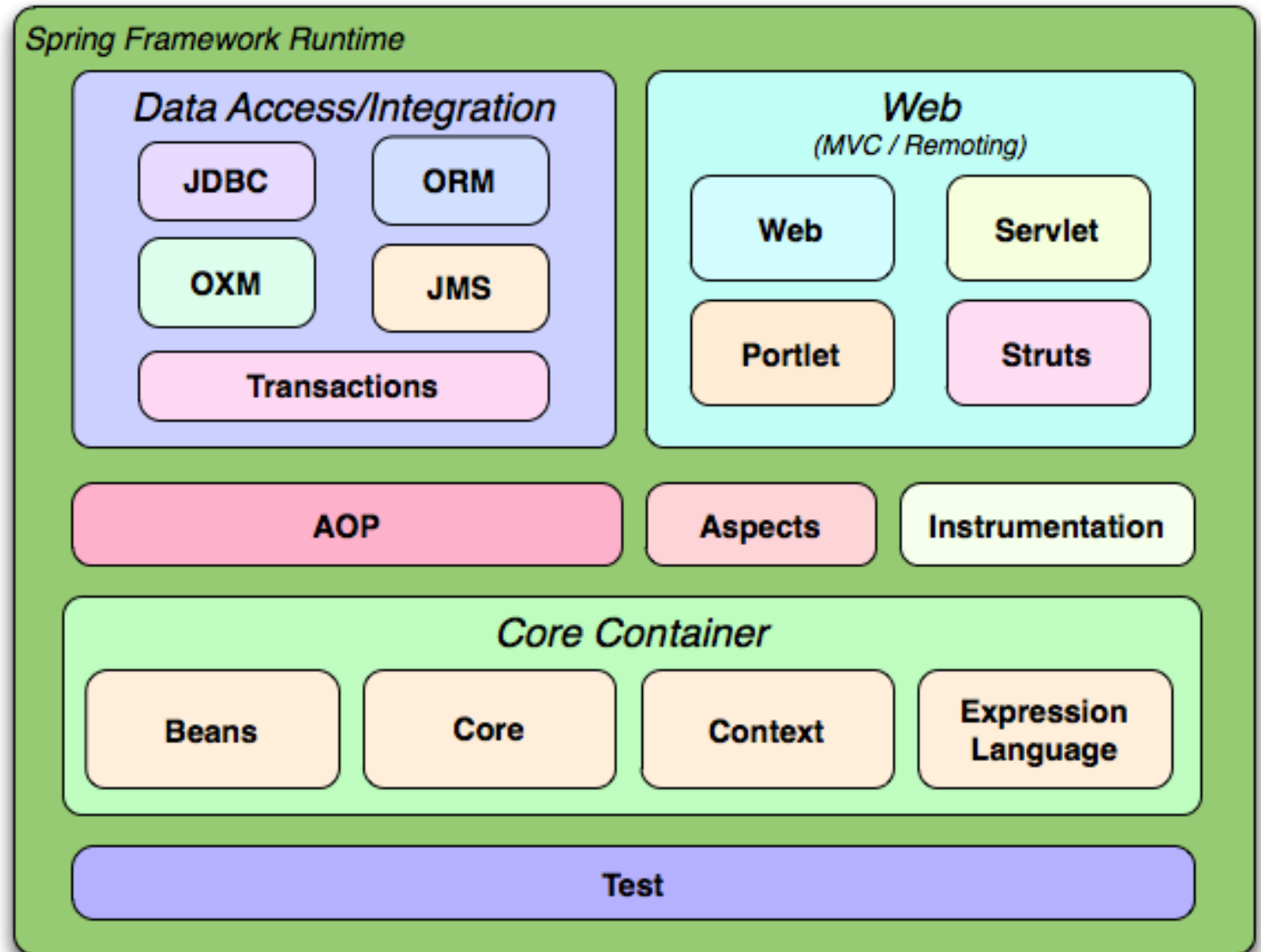
- Open-Source Framework
- Standalone and Enterprise application can be developed
- Released in 2003(initial), 2004(production) developed by Rod Johnson

Advantages

- Modular and lightweight(lightweight and easy to maintain applications)
- Flexible configuration(supports Java-based, XML-based and annotation-based configurations)
- Dependency Injection(dependency management)
- Aspect oriented programming(Allows developers to separate code from the features like logging, transactions, security etc.)
- Easy database handling(reduce boilerplate code increase efficiency)
- Testing support
- Security(robust framework for implementing authentication, authorization)
- High integration capabilities(with other frameworks and technologies like angular, react, JMS, SOAP, REST)
- High Scalability
- Open-Source

Modules

The Spring Framework consists of features organized into about 20 modules. These modules are grouped into Core Container, Data Access/Integration, Web, AOP (Aspect Oriented Programming), Instrumentation, and Test, as shown in the following diagram.



Dependency?

```
class Person {  
    int adhaar;  
    String Name;  
    Address address;  
}
```

```
class Address {  
    int streetNo;  
    String city;  
    String state;  
    Contact contact;  
}
```

```
class Contact {  
    double mob;  
    String email;  
}
```

- **Example of dependency**
- Code has very high degree of coupling due to aggregation
- To create Object of class Person, we depend on Address Object, and to create Address object, we need contact



Inversion of Control (IoC)

- Inversion of Control (IoC) is a **design principle** that emphasizes keeping Java classes independent of each other.
- IoC is achieved through **Dependency Injection (DI)**.
- IoC refers to transferring the control of objects and their dependencies from the main program to a container or framework.
- The IoC container uses two primary mechanisms to work:

Bean instantiation:

- The IoC container is responsible for creating and configuring beans. This can be done through XML configuration, Java annotations, or a combination of both.

Dependency injection:

- The IoC container injects dependencies into beans. This means that the IoC container is responsible for providing beans with the objects they need to function.

Spring Dependency Injection

- Dependency Injection (DI) is a **design pattern** that allows you to decouple your code by making it easier to create and manage objects.
- In Spring, DI is implemented using the Inversion of Control (IoC) container. The IoC container is responsible for creating and managing objects, and it injects them into your code when needed.
- Dependency Injection is a fundamental aspect of the Spring framework, through which the Spring container “*injects*” objects into other objects or “*dependencies*”.

There are three types of Spring Dependency Injection.

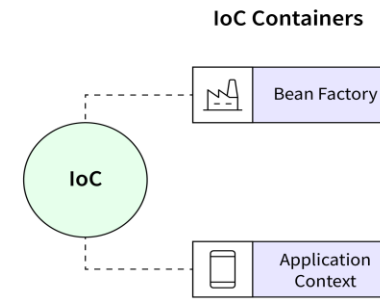
- **Setter Dependency Injection (SDI)**
- **Constructor Dependency Injection (CDI)**
- **Field Dependency Injection (FDI)**

Spring Container

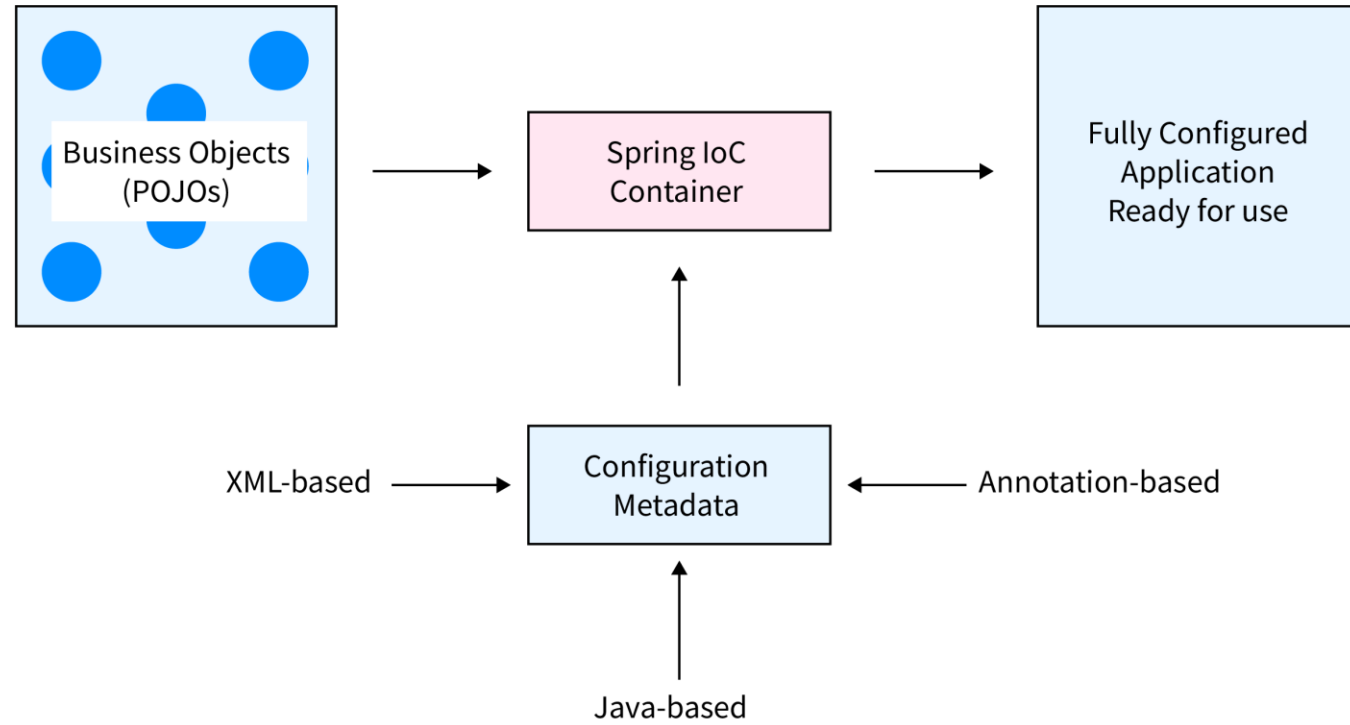
- The Spring container is the core of the Spring Framework.
- Manages Bean Objects(create, initialize, destroy)[Life cycle of bean]
- It is responsible for creating, configuring, and managing the objects that make up your application.
- The container uses a technique called dependency injection to manage the relationships between objects.
- Transaction Management

Spring container are of **TWO TYPES**

1. Bean factory(old Method)
2. ApplicationContext(**new Method**)



Spring IoC Container



| Spring IoC (Inversion of Control) | Spring Dependency Injection |
|---|--|
| Spring IoC Container is the core of Spring Framework. It creates the objects, configures and assembles their dependencies, manages their entire life cycle. | Spring Dependency injection is a way to inject the dependency of a framework component by the following ways of spring: Constructor Injection and Setter Injection |
| Spring helps in creating objects, managing objects, configurations, etc. because of IoC (Inversion of Control). | Spring framework helps in the creation of loosely-coupled applications because of Dependency Injection. |
| Spring IoC is achieved through Dependency Injection. | Dependency Injection is the method of providing the dependencies and Inversion of Control is the end result of Dependency Injection. |
| IoC is a design principle where the control flow of the program is inverted. | Dependency Injection is one of the subtypes of the IOC principle. |
| <u>Aspect-Oriented Programming</u> , <u>Dependency look up are other ways</u> to implement Inversion of Control. | In case of any changes in business requirements, no code change is required. |

Maven

Project build tool

Automates build
process



->Clean

->Compile

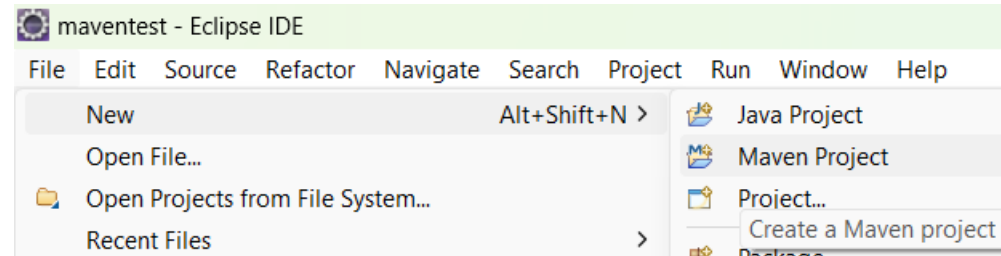
->test

->package

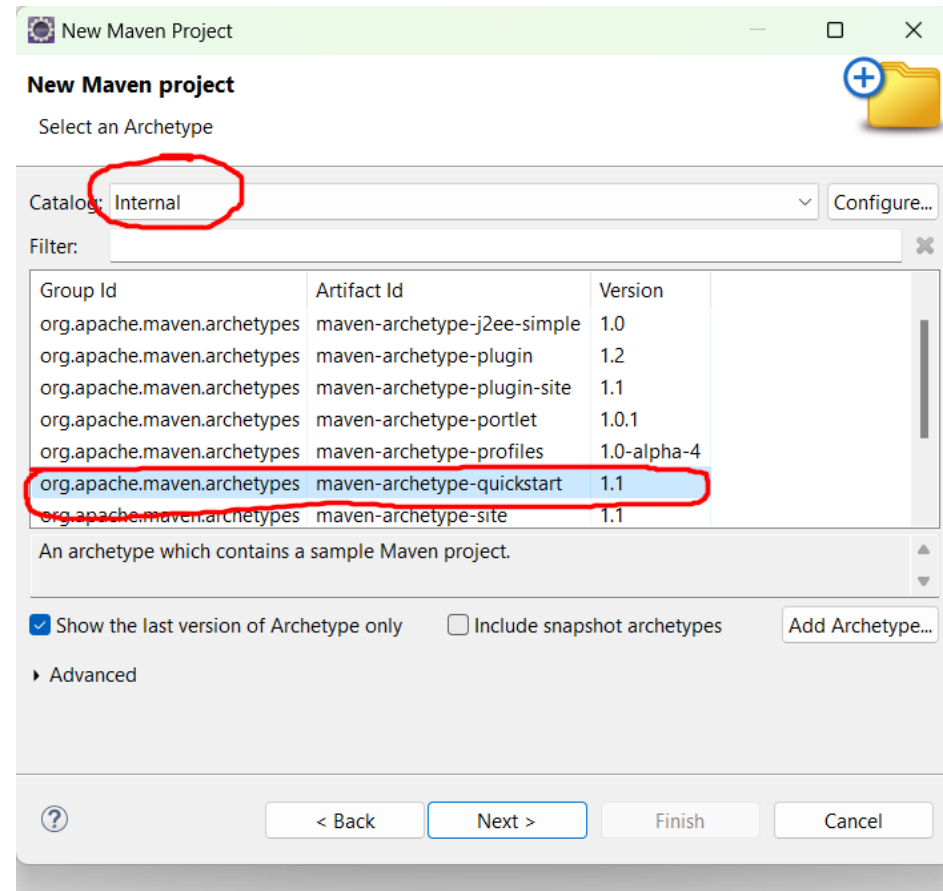
->install

->deploy

1. Create Maven project



2. Select Catalog and archetype as marked



3. Provide group id and artifact Id

4. Press finish

*(Internet must be on)

New Maven Project

New Maven project

Specify Archetype parameters

Group Id: com.test

Artifact Id: DIDemo

Version: 0.0.1-SNAPSHOT

Package: com.test.DIDemo

☒ run archetype generation interactively

Properties available from archetype:

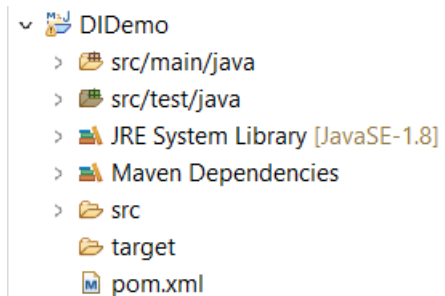
| Name | Value |
|------|-------|
| | |
| | |
| | |
| | |
| | |
| | |

Add... Remove

Advanced

? < Back Next > Finish Cancel

5. Open pom.xml(project object model)



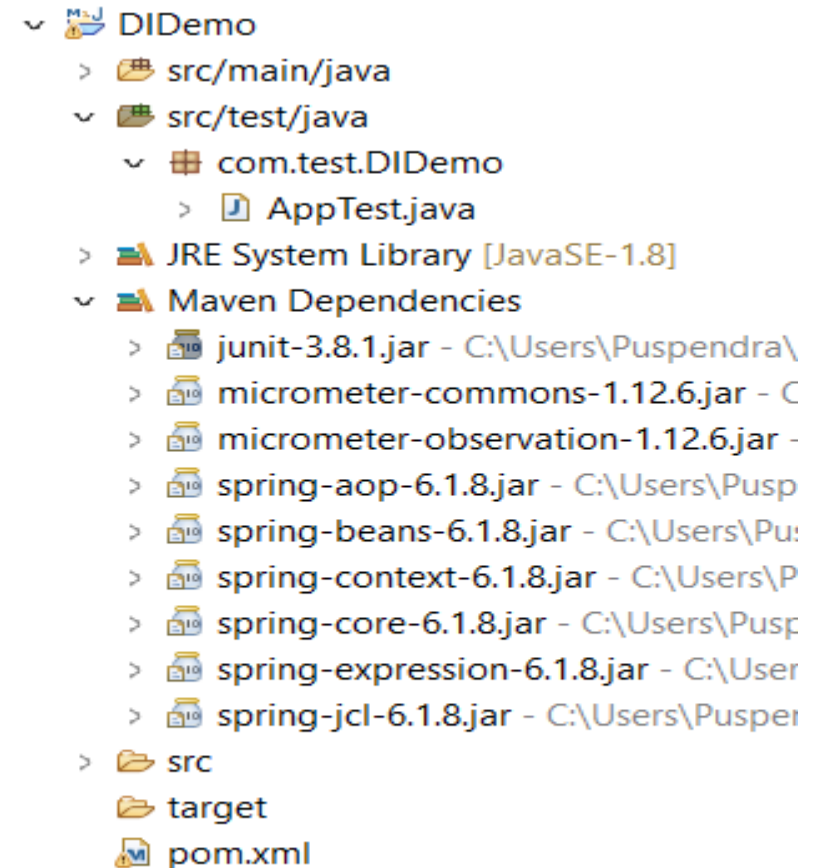
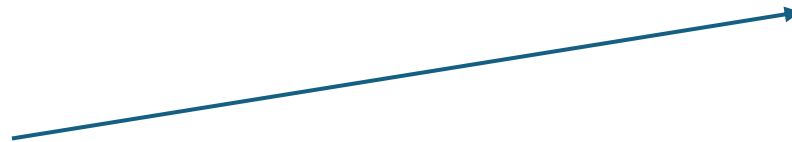
6. Add dependencies to <dependencies> tag

Press ctrl+s to save the pom.xml file(all dependencies will be downloaded automatically)

```
<dependency>  
<groupId>org.springframework</groupId>  
<artifactId>spring-core</artifactId>  
<version>6.1.8</version>  
</dependency>
```

```
<dependency>  
<groupId>org.springframework</groupId>  
<artifactId>spring-context</artifactId>  
<version>6.1.8</version>  
</dependency>
```

7. Check Project structure



8. Create new class Employee

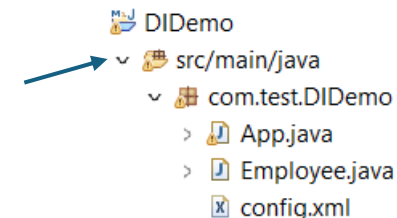
9. Create new xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
<bean class="com.test.DIDemo.Employee" name="stud1">
<property name="id" value="1" />
<property name="name" value="Raju" />
<property name="dept" value="Sales" />
</bean>
```

```
</beans>
```

```
package com.test.DIDemo;
public class Employee {
//POJO CLASS:FULLY ENCAPSULATED CLASS
private int empId;
private String name;
private String dept;
public Employee(int empId, String name, String dept) {
super();
this.empId = empId;
this.name = name;
this.dept = dept;
}
public int getEmpId() {
return empId;
}
public void setEmpId(int empId) {
this.empId = empId;
}
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
public String getDept() {
return dept;
}
public void setDept(String dept) {
this.dept = dept;
}
}
```

10. See project structure now




9. Update App.java

```
package com.test.DIDemo;  
import org.springframework.context.ApplicationContext;  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
public class App  
{  
    public static void main( String[] args )  
    {  
        System.out.println( "Hello World!" );  
        ApplicationContext context=new  
        ClassPathXmlApplicationContext("com/test/DIDemo/config.xml");  
        Employee employee=(Employee)context.getBean("stud1");  
        System.out.println(employee);  
    }  
}
```

Dependency injection



10. execute App.java



```
Problems  @ Javadoc  Declaration  Console ×  
<terminated> App (2) [Java Application] C:\Users\Puspendra\Downloads\eclipse-jee-2024-03  
Hello World!  
Employee [empId=1, name=Raju, dept=Sales]
```


Aspect-Oriented Programming (AOP)

- Aspect-Oriented Programming (AOP) is a programming technique that allows developers to modularize cross-cutting concerns. Cross-cutting concerns are tasks that affect multiple parts of a program, such as logging, security, and transaction management.
- AOP allows developers to separate these concerns from the main program logic. This makes the code more modular, reusable, and maintainable.
- Spring AOP is a popular implementation of AOP. It provides a simple and powerful way to write custom aspects.
- Spring provides simple and powerful ways of writing custom aspects by using either a schema-based approach or the @AspectJ annotation style. Both of these styles offer fully typed advice and use of the AspectJ pointcut language while still using Spring AOP for weaving.
- AOP is used in the Spring Framework to:
- Provide declarative enterprise services. The most important such service is declarative transaction management.
- Let users implement custom aspects, complementing their use of OOP with AOP.

Benefits of using AOP

Modularity:

- AOP allows developers to separate cross-cutting concerns from the main program logic. This makes the code more modular, reusable, and maintainable.

Reusability:

- Aspects can be reused across multiple projects. This saves time and effort, and it can help to improve the consistency of code.

Maintainability:

- AOP makes it easier to maintain code. This is because cross-cutting concerns are separated from the main program logic. This makes it easier to understand and modify the code.

BEAN SCOPE

- In the Spring Framework, a bean's scope determines how long it lives and how many instances of it are created.
- The default scope is *singleton*, which means that only one instance of the bean is created and shared across the entire application context. Other scopes include prototype, request, session, and global session.
- The *prototype scope* creates a new instance of the bean each time it is requested. This is useful for beans that are not thread-safe or that need to be customized for each request.
- The *request scope* creates a new instance of the bean for each HTTP request. This is useful for beans that need to be associated with a specific request, such as a database connection or a shopping cart.
- The *session scope* creates a new instance of the bean for each user session. This is useful for beans that need to be associated with a specific user, such as a user profile or a shopping cart.
- The *global session scope* creates a new instance of the bean for each user session across all applications in the same cluster. This is useful for beans that need to be shared across multiple applications, such as a user profile or a shopping cart.

BEAN SCOPE

- You can specify the scope of a bean using the *@Scope* annotation. For example, the following code creates a bean with the prototype scope:

```
@Scope("prototype")  
public class MyBean {  
    // ...  
}
```

OR

```
<bean class="com.test.DIDemo.Employee" name="stud1"  
scope="request">  
    <property...  
</bean>
```

WebSocket API

Spring Framework provides a WebSocket API that adapts to various WebSocket engines, including Tomcat, Jetty, GlassFish, WebLogic, and Undertow. This API allows developers to easily implement WebSocket-based applications. The Spring Framework also provides a number of features that make it easy to develop WebSocket-based applications, including:

- A messaging framework that supports STOMP, a text-oriented messaging protocol that can be used over any reliable 2-way streaming network protocol such as TCP and WebSocket.
- A JavaScript client library that makes it easy to develop WebSocket-based web applications.
- A number of pre-built WebSocket-based applications, such as a chat application and a stock ticker.

Autowiring

- **Autowiring** in the Spring framework can inject dependencies automatically.
- The Spring container detects those dependencies specified in the configuration file and the relationship between the beans. This is referred to as **Autowiring in Spring**.
- Autowiring in Spring internally uses constructor injection.
- An autowired application requires fewer lines of code comparatively but at the same time, it provides very little flexibility to the programmer.

| Modes | Description |
|---|--|
| No | This mode tells the framework that autowiring is not supposed to be done. It is the default mode used by Spring. |
| byName | It uses the name of the bean for injecting dependencies. |
| byType | It injects the dependency according to the type of bean. |
| Constructor | It injects the required dependencies by invoking the constructor. |
| Autodetect(depre- cated in Spring 3) | The autodetect mode uses two other modes for autowiring – constructor and byType. |

1. No

This mode tells the framework that autowiring is not supposed to be done. It is the default mode used by Spring.

2. byName

It uses the name of the bean for injecting dependencies. However, it requires that the name of the property and bean must be the same. It invokes the setter method internally for autowiring.

```
<bean id="state" class="pack.State">  
  <property name="name" value="UP" />  
</bean>  
<bean id="city" class="pack.City" autowire="byName"></bean>
```

3. byType

It injects the dependency according to the type of the bean. It looks up in the configuration file for the class type of the property. If it finds a bean that matches, it injects the property. If not, the program throws an error. The names of the property and bean can be different in this case. It invokes the setter method internally for autowiring.

```
<bean id="state" class="sample.State">  
  <property name="name" value="UP" />  
</bean>  
<bean id="city" class="sample.City" autowire="byType"></bean>
```


4. constructor

It injects the required dependencies by invoking the constructor. It works similar to the “byType” mode but it looks for the class type of the constructor arguments. If none or more than one bean are detected, then it throws an error, otherwise, it autowires the “byType” on all constructor arguments.

```
<bean id="state" class="sample.State">
  <property name="name" value="UP" />
</bean>
<bean id="city" class="sample.City" autowire="constructor"></bean>
```

5. autodetect

The autodetect mode uses two other modes for autowiring – constructor and byType. It first tries to autowire via the constructor mode and if it fails, it uses the byType mode for autowiring. It works in Spring 2.0 and 2.5 but is deprecated from Spring 3.0 onwards.

```
<bean id="state" class="sample.State">
  <property name="name" value="UP" />
</bean>
<bean id="city" class="sample.City" autowire="autodetect"></bean>
```

Auto wiring example

```
class City {
private int id;
private String name;
private State s;
public City() {
}
public int getID() { return id; }
public void setId(int eid) { this.id = eid; }
public String getName() { return name; }
public void setName(String st) { this.name = st; }
public State getS() {
return s;
}
public void setS(State s) {
this.s = s;
}
public int getId() {
return id;
}
public City(State s) {
super();
this.s = s;
}
public City(int id, String name, State s) {
super();
this.id = id;
this.name = name;
this.s = s;
}
@Override
public String toString() {
return "City [id=" + id + ", name=" + name + ", s=" + s + "];"
}
}
```

dependency

```
package com.test.DemoProject;
import org.springframework.beans.factory.annotation.Autowired;
public class State {
private String name;
public String getName() { return name; }
public void setName(String s) { this.name = s; }
@Override
public String toString() {
return "State [name=" + name + "];"
}
public State(String name) {
super();
this.name = name;
}
public State() {
super();}
}
```

//main

```
package com.test.DemoProject;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;
public class Test {
public static void main(String[] args) {
ApplicationContext context = new
ClassPathXmlApplicationContext("/com/test/DemoProject/config1.xml");
City city=context.getBean("city", City.class);
System.out.println(city);
}
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xmlns:p="http://www.springframework.org/schema/p"
xmlns:context="http://www.springframework.org/sche
ma/context"
xsi:schemaLocation="http://www.springframework.org
/schema/beans
http://www.springframework.org/schema/beans/spring
-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spri
ng-context.xsd">
<bean id="s" class="com.test.DemoProject.State" >
<property name="name" value="UP" />
</bean>
<bean name="city"
class="com.test.DemoProject.City"
autowire="constructor">
<property name="id" value="11" />
<property name="name" value="Washington, D.C." />
</bean>
</beans>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xmlns:p="http://www.springframework.org/schema/p"
xmlns:context="http://www.springframework.org/schema
/context"
xsi:schemaLocation="http://www.springframework.org/s
chema/beans
http://www.springframework.org/schema/beans/spring-
beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring
-context.xsd">
<bean id="s" class="com.test.DemoProject.State" >
<property name="name" value="UP" />
</bean>
<bean name="city" class="com.test.DemoProject.City"
autowire="byName">
<property name="id" value="11" />
<property name="name" value="Washington, D.C." />
</bean>
</beans>
```

@autowired

There are three ways to apply the @Autowired annotation:

NOTE: PUT FOLLOWING LINE IN CONFIG.XML FILE

```
<context:annotation-config/>
```

1. On a field: This is the most common way to use the @Autowired annotation. Simply annotate the field with @Autowired and Spring will inject an instance of the dependency into the field when the bean is created.

```
public class MyBean {  
    @Autowired  
    private MyDependency dependency;  
}
```

2. On a constructor: You can also use the @Autowired annotation on a constructor. This will cause Spring to inject an instance of the dependency into the constructor when the bean is created.

```
public class MyBean {  
    private MyDependency dependency;  
    @Autowired  
    public MyBean(MyDependency dependency) {  
        this.dependency = dependency;  
    }  
}
```

3. On a setter method: You can also use the @Autowired annotation on a setter method. This will cause Spring to inject an instance of the dependency into the setter method when the bean is created.

```
public class MyBean {  
    private MyDependency dependency;  
    @Autowired  
    public void setDependency(MyDependency dependency) {  
        this.dependency = dependency;  
    }  
}
```

The @Autowired annotation can be used on any field, constructor, or setter method that is declared in a Spring bean. The dependency that is injected must be a Spring bean itself.

Life Cycle Call backs

- Bean life cycle is managed by the spring container. When we run the program then, first of all, the spring container gets started. After that, the container creates the instance of a bean as per the request, and then dependencies are injected. And finally, the bean is destroyed when the spring container is closed. Therefore, if we want to execute some code on the bean instantiation and just after closing the spring container, then we can write that code inside the custom **init()** method and the **destroy()** method.



```
package com.kiet.FirstProject;
public class Employee {
    private int eid;
    private String name;
    public Employee() {
        super();
    }
    public Employee(int eid, String name) {
        super();
        this.eid = eid;
        this.name = name;
    }
    public int getId() {
        return eid;
    }
    public void setId(int eid) {
        this.eid = eid;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public void init()
    {
        System.out.println("initialized");
    }
    public void destroy()
    {
        System.out.println("destroyed");
    }
    @Override
    public String toString() {
        return "Employee [eid=" + eid + ", name=" + name + "]";
    }
}
```

```
<bean name="employee" class="com.kiet.FirstProject.Employee"
init-method="init" destroy-method="destroy">
<property name="eid" value="21"/>
<property name="name" value="Rohit Sharma"/>
</bean>
```

```
AbstractApplicationContext context=new
ClassPathXmlApplicationContext("/com/kiet/FirstProject/conf.xml");
Employee emp=context.getBean("employee",Employee.class);
context.registerShutdownHook();
System.out.println(emp);
```

Bean Configuration styles in spring framework

1. XML-based configuration: Traditional approach where beans are defined in XML configuration files (applicationContext.xml).

```
<bean name="address1" class="com.kiet.TEST.Address">  
<property name="city" value="Ghaziabad"/>  
<property name="state" value="UP"/>  
</bean>
```

2. Annotation-based configuration: Uses annotations like @Component, @Service, @Repository, and @Autowired for dependency injection and bean declaration.

Add following tag to config file.

```
<context:annotation-config/>  
<context:component-scan base-package="com.kiet.TEST"/>
```



```

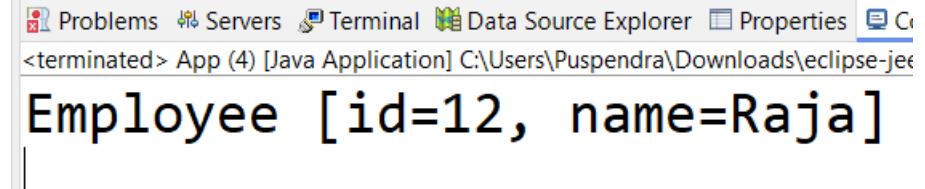
@Component
public class Employee {
    @Value("12")
    private int id;
    @Value("Raja")
    private String name;
    public Employee() {
    }
    public Employee(int id, String name) {
        super();
        this.id = id;
        this.name = name;
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    @Override
    public String toString() {
        return "Employee [id=" + id + ", name=" + name + "]";
    }
}

```

```

ApplicationContext context=new
ClassPathXmlApplicationContext("/com/kiet/TEST/configStereo.xml");
Employee emp=context.getBean("employee", Employee.class);
System.out.println(emp);

```



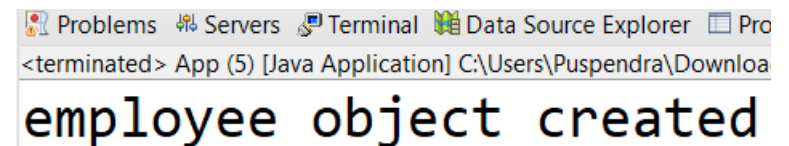
The screenshot shows the Eclipse IDE interface with several tabs: Problems, Servers, Terminal, Data Source Explorer, Properties, and Console. The Console tab is active, displaying the output of a Java application. The output text is "Employee [id=12, name=Raja]". Above the output, the console shows the command prompt "<terminated> App (4) [Java Application] C:\Users\Puspendra\Downloads\eclipse-jee".

3. Java-based Configuration (JavaConfig): Configuration is done using Java classes annotated with @Configuration. Beans are defined using @Bean annotations within these configuration classes.

```
@Component
public class Employee { //Employee.java
void doSomething()
{
System.out.println("employee object created");
}
}
```

```
//APP JAVA
ApplicationContext context=new AnnotationConfigApplicationContext(EmpConfig.class);
Employee emp=context.getBean("employee", Employee.class);
emp.doSomething();
```

```
@Configuration // EmpConfig.java
@ComponentScan(basePackages = "com.kiet.TESTa")
public class EmpConfig {
@Bean
public Employee getEmployee() {
return new Employee();
}
}
```



The screenshot shows an IDE interface with a terminal window at the bottom. The terminal title bar includes icons for Problems, Servers, Terminal, Data Source Explorer, and Project Explorer. The terminal text shows a command prompt followed by the output "employee object created".

SPRING BOOT

Spring Boot is an open-source Java framework that makes it easy to create Spring-based applications. It provides a number of features that make development faster and easier, including:

Auto-configuration:

- Spring Boot can automatically configure many of the beans that you need for your application, based on the dependencies that you have included. This means that you don't have to spend time writing a lot of XML configuration files.

Embedded servers:

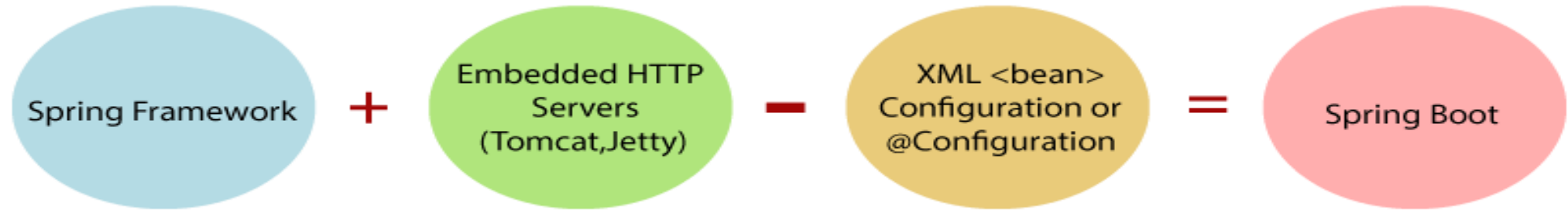
- Spring Boot can embed a number of different web servers, such as Tomcat and Jetty. This means that you don't have to deploy your application to a separate server.

Starter projects:

- Spring Boot provides a number of starter projects that you can use to get started quickly with different types of applications, such as web applications, RESTful web services, and batch applications.

Microservices:

- Spring Boot is a popular choice for developing microservices, which are small, independent services that can be combined to create a larger application. Spring Boot makes it easy to develop and deploy microservices, and it provides a number of features that are useful for microservices, such as embedded servers and service discovery.



Springboot Key Features

- Built on top of Spring Framework
- Simplifies development and deployment of Spring applications
- Auto-configuration
- Starter dependencies
- Opinionated defaults
- Embedded servers (Tomcat, Jetty, etc.)
- Production-ready features (metrics, health checks, etc.)

SPRING BOOT

Benefits of Spring Boot

- Faster development time
- Reduced boilerplate code
- Integrated development environment
- Easy deployment (self-contained JARs)
- Microservices architecture support

SPRING BOOT BUILD SYSTEMS

Spring Boot supports several build systems, but it primarily recommends two: **Maven and Gradle**

These build systems are recommended because they offer excellent support for dependency management.

- **Maven:** Maven uses an XML file (pom.xml) for its configuration. It provides a uniform build system and a quality project information. Spring Boot's spring-boot-starter-parent project can be used as a parent project in Maven to provide sensible defaults.
- **Gradle:** Gradle uses Groovy-based DSL (Domain Specific Language) for its configuration. It's known for its performance and flexibility.

| | Maven | Gradle |
|-----------------------|---|--|
| Based On | Maven is based on developing pure Java language-based software | Gradle, on the other hand, is based on developing domain-specific language projects |
| Configuration | Maven uses an XML file for its configuration and follows strict conventions. This makes it easy for developers to understand the layout of a project, but also limits customization options | Gradle uses a Groovy-based Domain-Specific Language (DSL) for its configuration, providing more flexibility in the build process |
| Performance | Maven, in terms of execution of projects, is quite slow | Gradle is known for its performance and is generally faster than Maven2. It uses mechanisms like incrementality, build cache, and the Gradle Daemon for work avoidance and faster builds |
| Flexibility | Maven provides a very rigid model that makes customization tedious and sometimes impossible. | Gradle is highly customizable and can be modified under various technologies for diverse projects |
| User Experience | Maven's longer tenure means that its support through IDEs is better for many users | Gradle's IDE support continues to improve quickly |
| Dependency Management | Both Gradle and Maven offer excellent support for dependency management. However, Gradle's mechanisms for work avoidance and incrementality make it much faster than Maven | |

Structuring Spring Boot Code

Spring Boot does ***not require any specific code*** layout to work. However, there are some best practices that help.

According to Spring boot documentation:

Place your ***main application class*** in a root package above other classes. The ***@EnableAutoConfiguration*** annotation is often placed on your main class, and it implicitly defines a base “search package” for certain items. For example, if you are writing a JPA application, the package of the ***@EnableAutoConfiguration*** annotated class is used to search for ***@Entity*** items.

Using a root package also lets the ***@ComponentScan*** annotation be used without needing to specify a basePackage attribute. You can also use the ***@SpringBootApplication*** annotation if your main class is in the root package.

The following listing shows a typical layout:

```
com
+- example
    +- myapplication
        +- Application.java
        |
        +- customer
            +- Customer.java
            +- CustomerController.java
            +- CustomerService.java
            +- CustomerRepository.java
            |
        +- order
            +- Order.java
            +- OrderController.java
            +- OrderService.java
            +- OrderRepository.java
```

Main class

```
package com.example.myapplication;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
```

@Configuration

@EnableAutoConfiguration

@ComponentScan

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

Spring Boot Runners

Spring Boot Runners are interfaces that allow you to execute code when a Spring Boot application starts. They are typically used to perform some initialization or setup tasks before the application starts processing requests.

There are two types of runners:

CommandLineRunner:

- This is the legacy runner that was introduced in Spring Boot 1.0. It has one abstract method, `run(String... args): void`.

ApplicationRunner:

- This is a newer runner that was introduced in Spring Boot 1.3. It has one abstract method, `run(ApplicationArguments args)` throws Exception.

The main ***difference*** between the two runners is that the ApplicationRunner gives you access to the application arguments, while the CommandLineRunner does not.

Logger

A logger in Spring Boot is a tool that helps developers track and monitor the events that happen in their application. It is a crucial part of any application, as it can help identify and fix errors, as well as track performance and usage.

Spring Boot comes with a built-in logger, which is based on the popular Logback logging framework. This logger can be used to log messages to the console, to a file, or to a remote server.

Benefits of using a logger in Spring Boot:

- Improved debugging
- Better performance monitoring
- Increased security
- Enhanced compliance

BASICS OF WEB APPLICATIONS

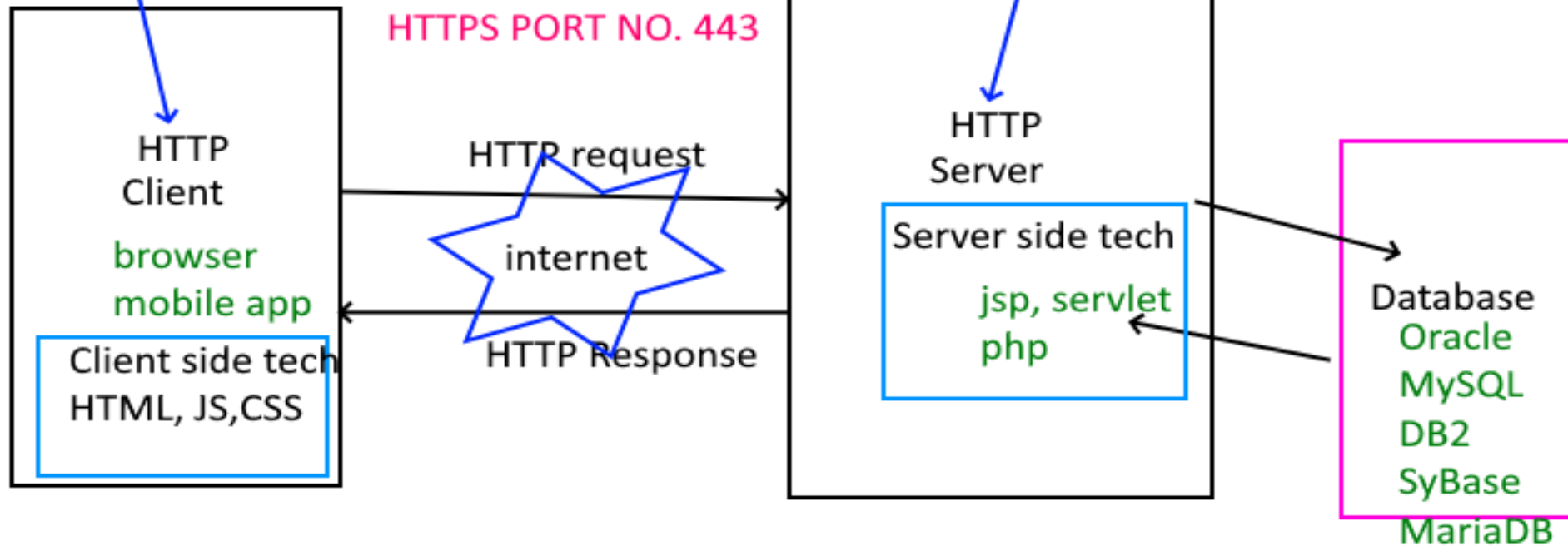
- **Localhost:** In a computer network, localhost refers to the computer you are using to send a loopback request. This request is data from your computer acting as a virtual server, which is sent back to itself without ever passing through a physical network.
- Server at localhost refers to your own machine is hosting(server) at “localhost” or <http://127.0.0.1>
- Spring Boot has inbuilt “tomcat-apache” server.(no external installation is required.

A program able to send
HTTP requests

A continuously running
program able to serve
HTTP requests

(web applications runs on)
HTTP Port no 80
HTTPS PORT NO. 443

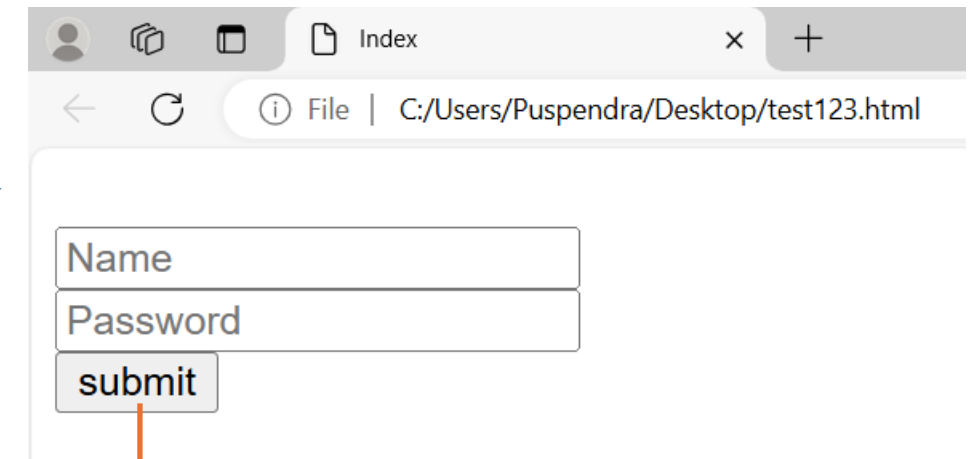
(TOMCAT,JETTY, GLASHFISH ETC)



```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Index</title>
</head>
<body>
<form action="#app" method="get"><br>
<input type="text" placeholder="Name"/><br>
<input type="password" placeholder="Password">
<input type="submit" value="submit">
</form>
</body>
</html>
```

Method can be get, post, put, delete, options etc

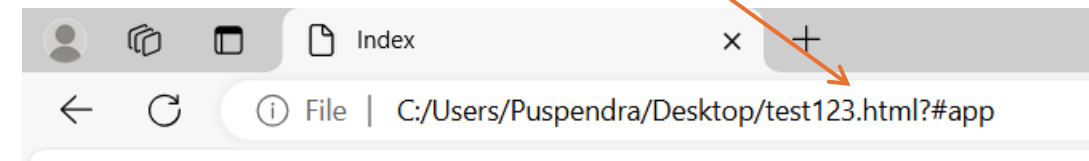
result



A screenshot of a web browser window. The address bar shows the file path C:/Users/Puspendra/Desktop/test123.html. The page content consists of a form with two input fields: 'Name' and 'Password', followed by a 'submit' button.

On pressing submit button

Url is appended with request
Parameters(get method)



A screenshot of a web browser window showing the result of a GET request. The address bar now displays the URL C:/Users/Puspendra/Desktop/test123.html?#app, indicating that the form data has been appended to the URL as a query string.

GET, POST, PUT, and DELETE are HTTP methods that define how clients and servers communicate over the World Wide Web. HTTP methods enable API clients to perform CRUD (Create, Read, Update, and Delete) actions on an API's resources in a standardized and predictable way. The most commonly used HTTP methods are:

GET

The GET method is used to retrieve data on a server. Clients can use the GET method to access all of the resources of a given type, or they can use it to access a specific resource. For instance, a GET request to the /products endpoint of an e-commerce API would return all of the products in the database, while a GET request to the /products/123 endpoint would return the specific product with an ID of 123. GET requests typically do not include a request body, as the client is not attempting to create or update data.

POST

The POST method is used to create new resources. For instance, if the manager of an e-commerce store wanted to add a new product to the database, they would send a POST request to the /products endpoint. Unlike GET requests, POST requests typically include a request body, which is where the client specifies the attributes of the resource to be created. For example, a POST request to the /products endpoint might have a request body that looks like this:

```
{  
  "name": "Sneakers",  
  "color": "blue",  
  "price": 59.95,  
  "currency": "USD"  
}
```


PUT

The PUT method is used to replace an existing resource with an updated version. This method works by replacing the entire resource (i.e., the specific product located at the `/products/123` endpoint) with the data that is included in the request's body. This means that any fields or properties not included in the request body are deleted, and any new fields or properties are added.

PATCH

The PATCH method is used to update an existing resource. It is similar to PUT, except that PATCH enables clients to update specific properties on a resource—without overwriting the others. For instance, if you have a product resource with fields for name, brand, and price, but you only want to update the price, you could use the PATCH method to send a request that only includes the new value for the price field. The rest of the resource would remain unchanged. This behavior makes the PATCH method more flexible and efficient than PUT.

DELETE

The DELETE method is used to remove data from a database. When a client sends a DELETE request, it is requesting that the resource at the specified URL be removed. For example, a DELETE request to the `/products/123` endpoint will permanently remove the product with an ID of 123 from the database. Some APIs may leverage authorization mechanisms to ensure that only clients with the appropriate permissions are able to delete resources.

Creating “hello world” application in Spring Boot

Step 1: <https://start.spring.io/>



Project

☐ Gradle - Groovy ☐ Gradle - Kotlin ☒ Java ☐ Kotlin ☐ Groovy
☒ Maven

Spring Boot

☐ 3.3.2 (SNAPSHOT) ☒ 3.3.1 ☐ 3.2.8 (SNAPSHOT) ☐ 3.2.7

Project Metadata

Group

Artifact

Name

Description

Package name

Packaging ☒ Jar ☐ War

Java ☐ 22 ☐ 21 ☒ 17

Dependencies

[ADD DEPENDENCIES...](#) CTRL + B

No dependency selected

[GENERATE](#) CTRL + G

[EXPLORE](#) CTRL + SPACE

[SHARE...](#)

Step 2:

Dependencies → add Web

Language → Java

Build → Maven

Group id → com.myorganization

Artifact → SpringBootDemo

Step 3:

Press Generate

Step 4:

Unzip downloaded project

Dependencies

ADD DEPENDENCIES... CTRL + B

Spring Web

WEB

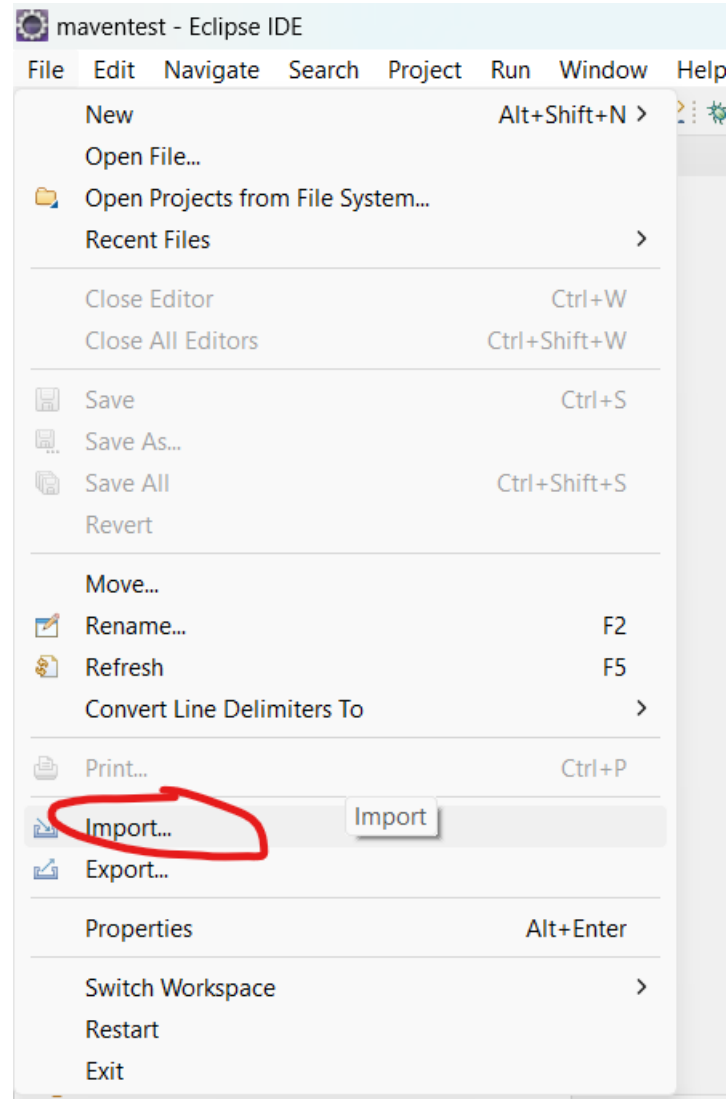
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

GENERATE CTRL + ⌘

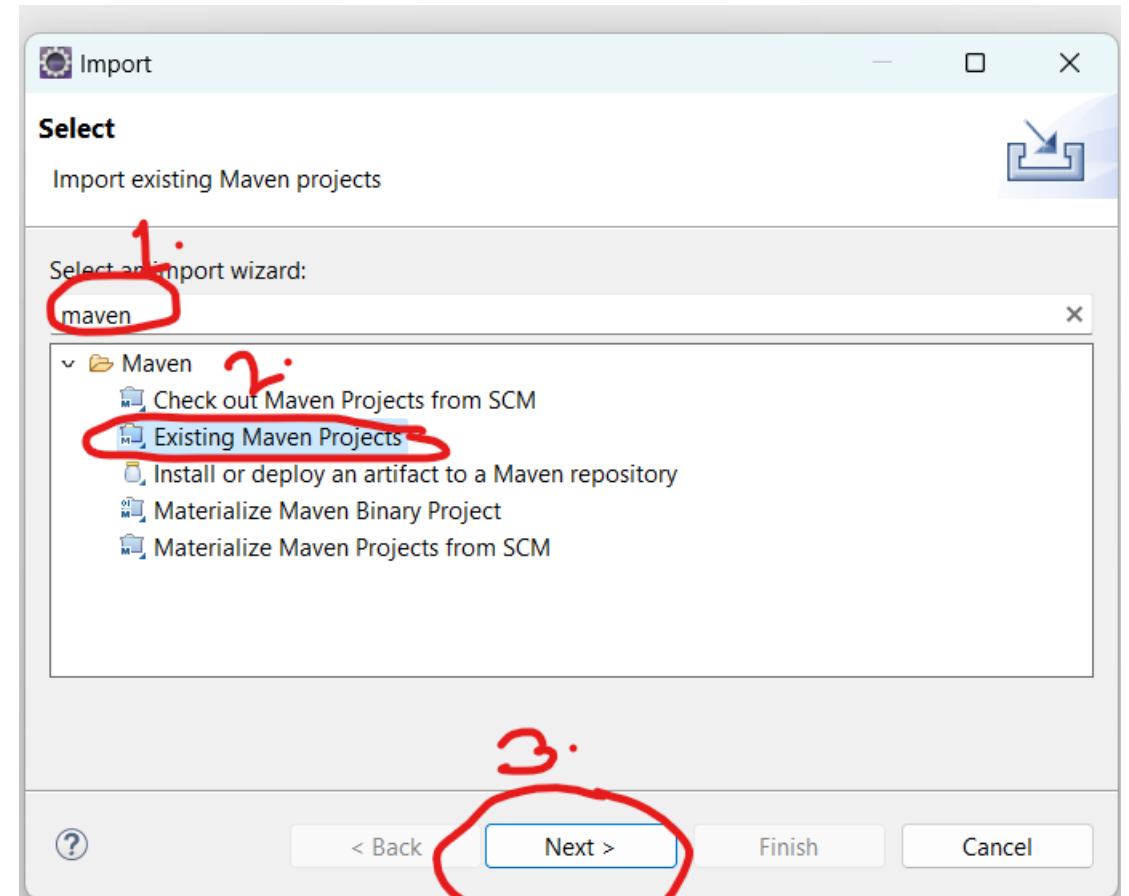
EXPLORE CTRL + SPACE

SHARE...

Step 5:
click
import

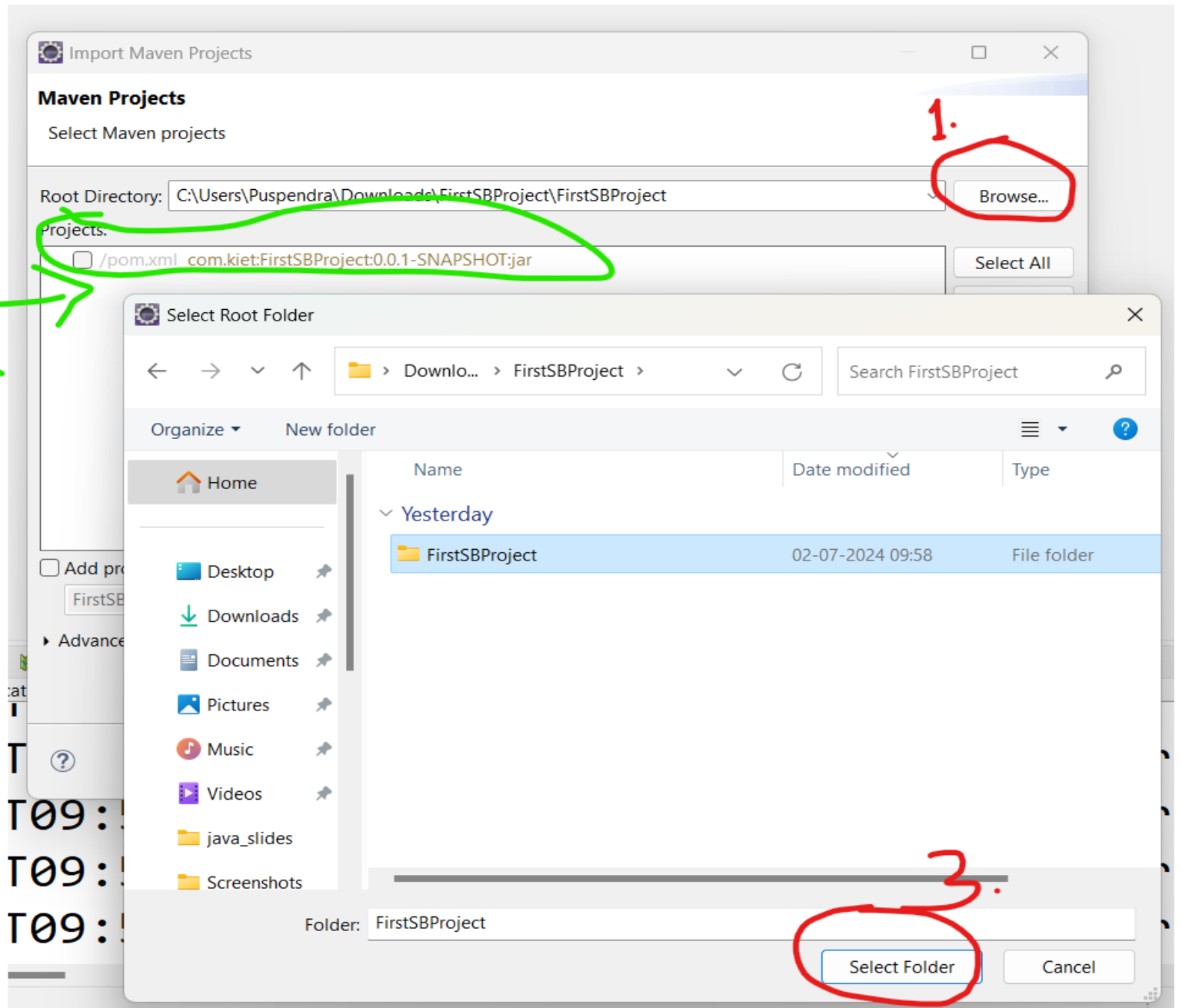


Step 6:
click
next

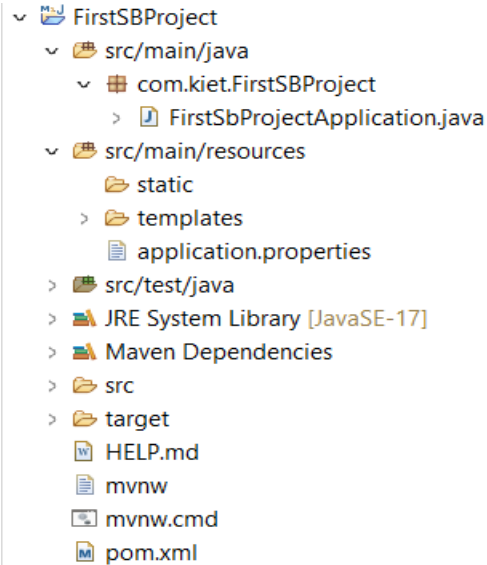


Step 7:
click
Select
folder

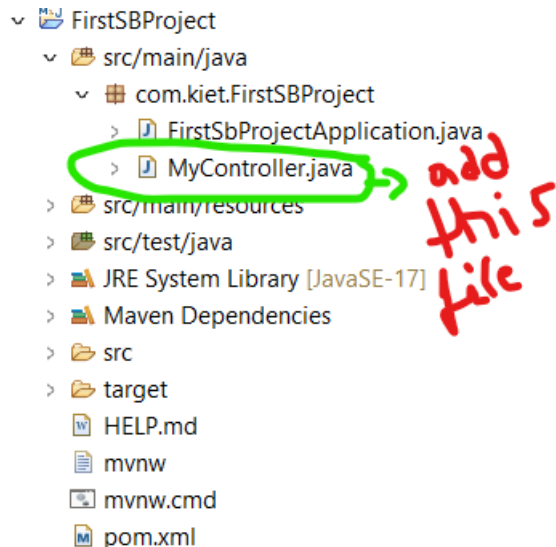
2. Check



Step 8: check application



Step 9: Add controller



Step 10: Create end points in controller

```
package com.kiet.FirstSBProject;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;
```

@Controller

```
public class MyController {
```

@GetMapping("/hello")//END POINT -1

@ResponseBody

```
public String hello(@RequestParam(value = "name",
defaultValue = "World") String name) {
    return String.format("Hello %s!", name);
}
```

@GetMapping("/hi")//END POINT -2

@ResponseBody

```
public String hello() {
    return "hi";
}
}
```

Step 11: run main

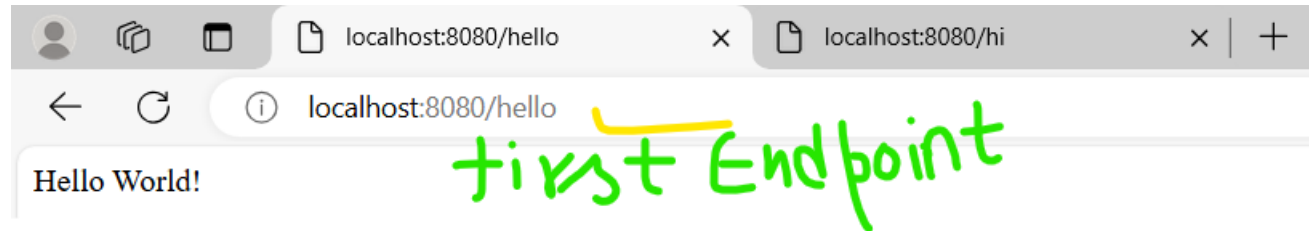
Step 12: see logs

[illegible]

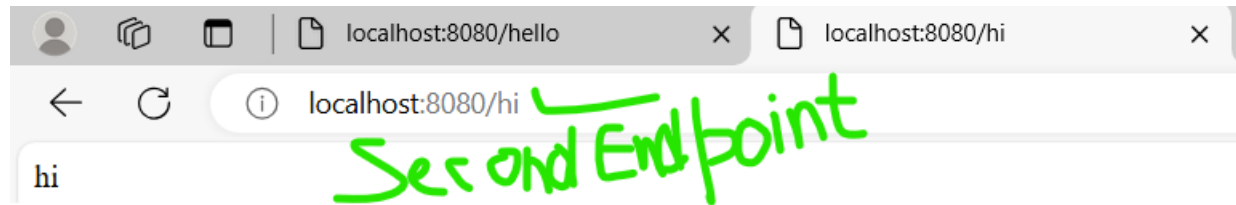
```
:: Spring Boot ::                (v3.3.1)
```

```
2024-07-04T11:30:02.466+05:30 INFO 6592 --- [FirstSBProject] [
2024-07-04T11:30:02.472+05:30 INFO 6592 --- [FirstSBProject] [
```

Step 13: test 1



Step 14: test 2



Spring Boot Starters

Spring Boot Starters are a set of convenient dependency descriptors that you can include in your application. They help you get a project up and running quickly by providing a collection of curated dependencies that are commonly used together.

Commonly Used Starters

spring-boot-starter-web

- Purpose:** For building web applications using Spring MVC.
- Includes:** Spring MVC, embedded Tomcat/Jetty/Undertow server, and necessary dependencies for web development.

spring-boot-starter-data-jpa

Purpose: For using Spring Data JPA with Hibernate for data persistence.

Includes: Spring Data JPA, Hibernate, JDBC, and necessary dependencies for database access and ORM.

spring-boot-starter-security

Purpose: For adding security to Spring Boot applications.

Includes: Spring Security, OAuth, and other security-related dependencies.

RESTFUL WEB SERVICES

RESTful web services are a type of web service that uses the HTTP protocol to communicate with clients. They are based on the Representational State Transfer (REST) architectural style, which defines a set of constraints that make web services more interoperable and scalable.

REST Controller

- A REST controller is a type of controller that is used to create RESTful web services. REST stands for REpresentational State Transfer, and it is an architectural style for designing web services.
- REST controllers are typically annotated with the `@RestController` annotation. This annotation tells Spring that the controller is a REST controller, and it enables a number of features that are useful for developing RESTful web services, such as automatic serialization of responses to JSON or XML.

Difference between Controller and Restcontroller



```
@RestController
public class MyController {
    @GetMapping("/hello")
    public String hello() {
        return "Hello, world!";
    }
}
```

Request Mapping:

- Request Mapping is an annotation that is used to map web requests to specific handler classes and/or handler methods.
- The annotation is used at the class level to express shared mappings or at the method level to narrow down to a specific endpoint mapping.
- The annotation has various attributes to match by URL, HTTP method, request parameters, headers, and media types.

Request Body:

- A request body is data sent by the client to your API.
- It is typically used when we expect the client to send data in the request body when submitting a form or sending JSON data.
- You can use the `@RequestBody` annotation in Spring Boot to bind the HTTP request body to a parameter in a controller method.

```
@RestController
@RequestMapping("/student")
public class StudentController {
    StudentService studentService;
    public StudentController(StudentService
studentService) {
        super();
        this.studentService = studentService;
    }
    @PostMapping
    Student addStudent(@RequestBody Student
student)
    {
        return
        this.studentService.addStudent(student);
    }
}
```

Path Variable:

- Path variables are used to extract values from the URI and use them as method parameters.
- You can use the `@PathVariable` annotation in Spring Boot to extract values from the URI path.
- Path variables are typically used to handle dynamic URIs where one or more of the URI value works as a parameter.

```
@GetMapping("/{id}")
Student getStudent(@PathVariable int id)
{
    return this.studentService.getStudent(id);
}
```

Request Parameter:

- Request parameters are used to extract input data passed through an HTTP URL or passed as a query.
- You can use the `@RequestParam` annotation in Spring Boot to extract input data passed through an HTTP URL or passed as a query.
- Request parameters are typically used to filter data or to pass additional information to the controller method.

```
@ResponseBody
public String hello(@RequestParam(value =
    "name", defaultValue = "World") String name) {
    return String.format("Hello %s!", name);
}
```

GET, POST, PUT, and DELETE

GET, POST, PUT, and DELETE are the four most common **HTTP methods** used in Spring Boot.

- GET: is used to retrieve data from a server.
- POST: is used to create new data on a server.
- PUT: is used to update existing data on a server.
- DELETE: is used to delete data from a server.
- In Spring Boot, these methods are mapped to controller methods using the **@GetMapping**, **@PostMapping**, **@PutMapping**, and **@DeleteMapping** annotations, respectively.