Palash Bajpai

Spring Boot

Table of Contents

[**1. Introduction to Spring Boot** 2](#_Toc202017684)

[🔹 **Important Features of Spring Boot** 2](#_Toc202017685)

[🔹 **Spring Boot Vs Spring** 3](#_Toc202017686)

[🔹 **Hello World API** 4](#_Toc202017687)

[**2. Web Service** 6](#_Toc202017688)

[🔹 **Types of Web Services** 7](#_Toc202017689)

[**3. REST API with Spring Boot** 9](#_Toc202017690)

[🔹 **Swagger Documentation** 15](#_Toc202017691)

[🔹 **Spring Boot Actuators** 16](#_Toc202017692)

[**4. Spring JPA** 17](#_Toc202017693)

[🔹 **JPQL(Java Persistence Query Language)** 22](#_Toc202017694)

[🔹 **Derived Query Methods** 23](#_Toc202017695)

[**5. Spring Configuration and profiling** 24](#_Toc202017696)

[🔹 **application.properties and application.yml** 24](#_Toc202017697)

[🔹 **External Configuration** 25](#_Toc202017698)

[🔹 **Profiles** 26](#_Toc202017699)

[**6. Spring AOP** 28](#_Toc202017700)

[**🔹 Core AOP Concepts** 28](#_Toc202017701)

[**🔹 SLF4J** 32](#_Toc202017702)

# **1. Introduction to Spring Boot**

*Spring Boot is an opinionated framework built on top of the Spring Framework to simplify application development by:*

* Eliminating boilerplate configurations
* Providing default behaviors (auto-configuration)
* Embedding web servers (like Tomcat)
* Enabling production-ready features out-of-the-box
* Focuses on **convention over configuration**.

"**Opinionated**" means Spring Boot makes smart decisions for you (e.g., which configurations to use) so you can get started quickly and only override when needed.

## 🔹 **Important Features of Spring Boot**

**1. Auto-Configuration:**

Based on classpath scanning, it configures your app automatically. Behind the scenes, it uses @EnableAutoConfiguration and conditionals like @ConditionalOnClass, @ConditionalOnMissingBean.

Example: If you have spring-boot-starter-data-jpa and H2 in your classpath, it automatically configures: DataSource, EntityManager, Hibernate dialect ,H2 Console

**@SpringBootApplication = @Configuration + @EnableAutoConfiguration + @ComponentScan**

Reads from:

* META-INF/spring.factories
* Checks conditions using @Conditional\* annotations

**2. Embedded Web Servers:**

Runs apps directly using embedded servers. No need for WAR or external server deployment.

**3. Spring Initializer:**

Scaffold a new project instantly: Visit: <https://start.spring.io/> . Instantly sets up project with correct directory structure and config.

**4. Starter Dependencies:**

Reduces complexity by bundling compatible dependencies together.

<!--Instead of this-->  
<dependency>spring-core</dependency>  
<dependency>spring-context</dependency>  
<dependency>spring-webmvc</dependency>

<!--Use this:-->  
<dependency>  
<groupId>org.springframework.boot</groupId>  
<artifactId>spring-boot-starter-web</artifactId>  
</dependency>

**5. Spring Boot Actuator:**

Enables health checks, metrics, env info etc.

**6. Spring Boot DevTools:**

Live reload and automatic restarts during development.

**7. External Configuration via application.yml or application.properties:**

Supports multiple environments like:

# application-dev.yml  
server:  
 port: 8081  
spring:  
 datasource:  
 url: jdbc:h2:mem:testdb

## 🔹 **Spring Boot Vs Spring**

| **Feature** | **Spring Framework** | **Spring Boot** |
| --- | --- | --- |
| Setup | Manual (XML or Java) | Auto-config via starters |
| Web Server | External (Tomcat) | Embedded |
| Deployment | WAR file | Executable JAR |
| Config | Verbose, scattered | Centralized (application.yml) |
| Dependency Management | Manual | Starter POMs |
| Monitoring Tools | Custom setup | Built-in via Actuator |
| Learning Curve | Steep | Gentle (opinionated defaults) |
| Productivity | Medium | Very high |

## 🔹 **Hello World API**

*CODE EXAMPLE:*

1. Got to Spring initializer and initiate the project

Go to: <https://start.spring.io> .

Add spring boot web dependency:

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
</dependency>

spring-boot-starter-web: Includes everything for a web app:

* Spring MVC
* Embedded Tomcat
* Jackson (for JSON)
* Logging, etc.

We get this code:

@SpringBootApplication  
public class StudentManagementApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(StudentManagementApplication.class, args);  
 }  
  
}

@SpringBootApplication is a meta-annotation:

* Combines @Configuration, @EnableAutoConfiguration, and @ComponentScan.

SpringApplication.run(...) bootstraps the app:

* Starts the embedded Tomcat server
* Scans the classpath
* Initializes Spring context

2. Creating Hello world controller

import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.RestController;  
  
@RestController  
public class HelloController {  
  
 @GetMapping("/hello")  
 public String hello(){  
 return "Hello World";  
 }  
}

@RestController = @Controller + @ResponseBody

* Makes the method return JSON/text directly instead of rendering a view.

@GetMapping("/hello")

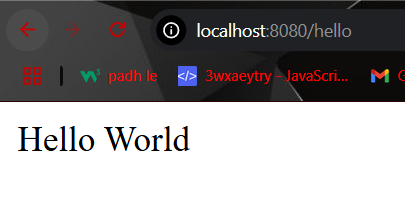
* Maps the HTTP GET request to /hello path.
* The method returns a plain text "Hello, World!".

3. Can change the port if needed: by changing in application.properties

spring.application.name=StudentManagement  
# Server port (default is 8080)  
server.port=8080

4. Output

Now go to: <http://localhost:8080/hello>



| **Component** | **Role** |
| --- | --- |
| @SpringBootApplication | Marks the main Spring Boot app class and sets up context |
| SpringApplication.run | Starts the app and embedded server |
| @RestController | Indicates a REST API controller |
| @GetMapping | Maps GET requests to methods |
| application.properties | Used for configuration like ports, paths, etc. |
| Embedded Tomcat | Auto-configured HTTP server |

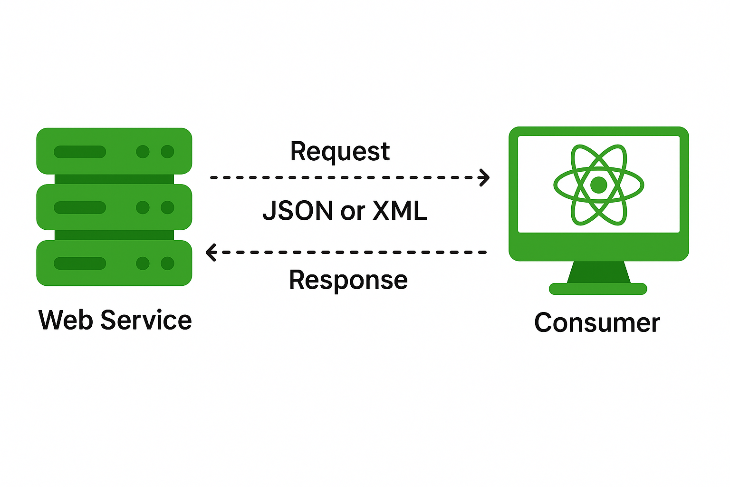
# **2. Web Service**

*A web service is a standardized way of enabling communication and data exchange between two software applications over a network (typically the Internet). These applications can be written in different programming languages and run on different platforms. Web services use protocols and standards to facilitate interoperability.*

* Eliminating boilerplate configurations
* Providing default behaviors (auto-configuration)
* Embedding web servers (like Tomcat)
* Enabling production-ready features out-of-the-box

**Key Characteristics:**

* **Platform-Independent:** Web services enable applications written in different languages (e.g., Java, Python, .NET) to communicate seamlessly. Also communication should be platform-independent (eg XML, Json).
* **Protocol-Based Communication:** Allows communication over a network and uses protocols like HTTP, HTTPS, and SOAP for exchanging data.
* **Loosely Coupled:** The client and server are not tightly bound; they communicate through interfaces defined by the service.
* **Interoperability:** Web services ensure interoperability across systems with different architectures.



**Key Components:**

* **Request:** Input to a web service
* **Response:** Output from the web service
* **Message Exchange Format (Payload Format):** XML or JSON for exchanging data.
* **Endpoints:** URL where the web services are accessible.
* **Service Provider (Server):** Entity providing the Web Service.
* **Service Consumer:** Entity consuming the Web Service.

## 🔹 **Types of Web Services**

**1. Soap ( Simple Object Access Protocol)**

SOAP is a protocol for exchanging structured information using XML over HTTP, SMTP, or other transport protocols.

**Features:**

* Strict standards for message format.
* Built-in error handling.
* Provides higher security (WS-Security).

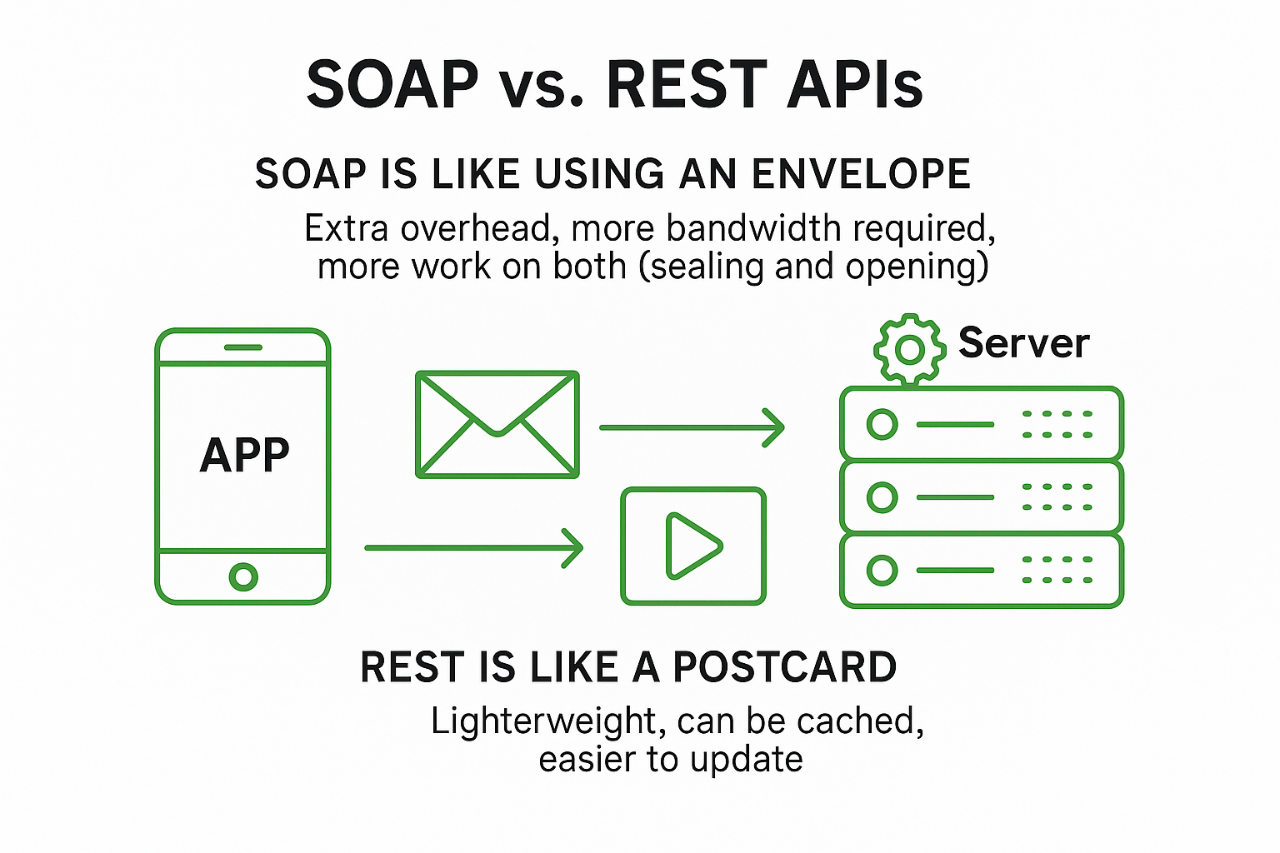
**Advantages:**

* Ideal for enterprise-level applications requiring reliability.
* Supports multiple transport protocols.

**Disadvantages:**

* Verbose and complex to implement.
* Slower due to XML-based payload.

**Example:** A banking service providing secure account details.



**2. REST (Representational State Transfer) Web Services**

REST is an architectural style for designing lightweight web services using standard HTTP methods.

**Features:**

* Stateless communication.
* Supports multiple formats (XML, JSON, etc.).

**Advantages:**

* Lightweight and faster.
* Easier to integrate with modern web applications.
* Broadly used for APIs.

**Disadvantages:**

* Lacks built-in security mechanisms (depends on HTTPS).

**Example:** A REST API for retrieving weather data.

| **Feature** | **REST (Representational State Transfer)** | **SOAP (Simple Object Access Protocol)** |
| --- | --- | --- |
| **Protocol** | Typically uses **HTTP** | Uses **any protocol** (HTTP, SMTP, TCP, etc.) |
| **Message Format** | Usually **JSON** (can also use XML, HTML, plain text) | Strictly **XML** |
| **Interface Style** | **Resource-based** (URLs represent resources) | **Operation-based** (uses services and functions) |
| **Flexibility** | Lightweight and flexible | Heavy and rigid |
| **Performance** | Generally **faster** and **uses less bandwidth** | Slower due to XML parsing and additional headers |
| **Standards Compliance** | Less standardized (no strict rules) | Highly standardized (WSDL, XSD, WS-Security, etc.) |
| **Ease of Use** | Easy to use and test via browser or tools like Postman | Harder to test without dedicated tools like SoapUI |
| **Security** | Relies on HTTPS and OAuth for security | Built-in support for **WS-Security** (encryption, signing) |
| **Error Handling** | HTTP status codes (e.g., 404, 500) | Uses **SOAP Fault** messages |
| **Data Types Support** | Limited to HTTP-friendly types (text, JSON, XML, etc.) | Supports **complex data types** via XML Schema |
| **Tooling Support** | Easy integration with modern web tools (React, Angular, etc.) | Strong support in enterprise environments and legacy tools |
| **Service Description** | Can use **OpenAPI/Swagger** for documentation | Uses **WSDL** (Web Services Description Language) |
| **Caching** | Supports caching (using HTTP GET) | No inherent caching mechanism |
| **Use Cases** | Web/mobile APIs, microservices, public APIs | Enterprise systems, banking, payment gateways |

# **3. REST API with Spring Boot**

A RESTful API is an API that adheres to REST constraints, making it stateless, cacheable, layered, and uniform.

**Key REST Concepts:**

* **Resource**: Anything that can be named (e.g., a User, Order, etc.)
* **URI**: Identifies a resource (e.g., /api/users/1)
* **Representation**: JSON or XML structure sent to the client
* **Statelessness**: No session is stored on the server
* **HTTP Verbs**: Mapped to CRUD operations
  + GET → Read
  + POST → Create
  + PUT → Update
  + DELETE → Delete

***CODE EXAMPLE:***

1. We will create Student Management System supporting CRUD operation

Go to: <https://start.spring.io> .

Add spring boot web dependency:

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
</dependency>

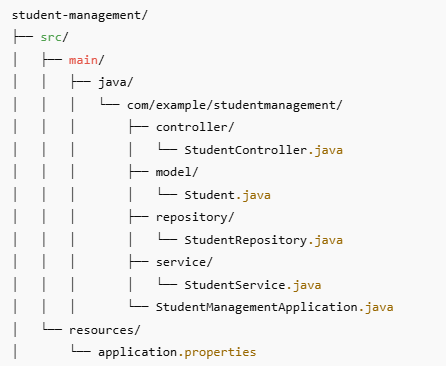
<dependency>  
 <groupId>org.projectlombok</groupId>  
 <artifactId>lombok</artifactId>  
 <optional>true</optional>  
</dependency>

**Project Lombok** is a **Java library** that helps developers eliminate boilerplate code like Getters and Setters, Constructors, equals() and hashCode(), toString(), Builders etc.

It does this using **annotations** that generate code **at compile time**, making your Java classes cleaner and easier to maintain.

| **Annotation** | **One-liner Explanation** |
| --- | --- |
| **@Getter** | Generates getter methods for all or selected fields. |
| **@Setter** | Generates setter methods for all or selected fields. |
| **@ToString** | Automatically generates a toString() method. |
| **@EqualsAndHashCode** | Generates equals() and hashCode() methods based on fields. |
| **@Data** | A shortcut for @Getter, @Setter, @ToString, @EqualsAndHashCode, and @RequiredArgsConstructor. |
| **@NoArgsConstructor** | Generates a no-argument constructor. |
| **@AllArgsConstructor** | Generates a constructor with parameters for all fields. |
| **@RequiredArgsConstructor** | Generates a constructor for final and @NonNull fields only. |
| **@Builder** | Implements the builder pattern for easy object construction. |
| **@Value** | Makes the class immutable (final fields, private constructor, only getters). |
| **@Slf4j** | Adds a Logger object named log using SLF4J logging API. |
| **@NonNull** | Generates a null-check for the parameter in constructors or methods. |

Project Structure



2. Basic File which initiate SpringBootApplication

package com.example.studentManagement;  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
  
@SpringBootApplication // Combines @Configuration, @EnableAutoConfiguration, and @ComponentScan

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

**@SpringBootApplication**: Convenience annotation that auto-configures Spring Boot app and scans for components in the same package or sub-packages.

3. Creating Student.java file in model

package com.example.studentManagement.model;  
  
public class Student {  
  
 private Long id;  
 private String name;  
 private String email;  
 private int age;  
  
 // Constructors  
 public Student() {}  
  
 public Student(Long id, String name, String email, int age) {  
 this.id = id;  
 this.name = name;  
 this.email = email;  
 this.age = age;  
 }  
  
 // Getters and Setters  
 public Long getId() { return id; }  
 public void setId(Long id) { this.id = id; }  
  
 public String getName() { return name; }  
 public void setName(String name) { this.name = name; }  
  
 public String getEmail() { return email; }  
 public void setEmail(String email) { this.email = email; }  
  
 public int getAge() { return age; }  
 public void setAge(int age) { this.age = age; }  
}

Or Can use Lombok for same, sometimes Lombok gives issues in intellij. Remember to install Lombok plugin in intellij while using it.

package com.example.studentManagement.model;  
  
import lombok.AllArgsConstructor;  
import lombok.Data;  
import lombok.NoArgsConstructor;  
  
@Data // Generates getters, setters, toString, equals, and hashCode  
@NoArgsConstructor // Generates a no-argument constructor  
@AllArgsConstructor // Generates an all-arguments constructor  
public class Student {  
  
 private Long id;  
 private String name;  
 private String email;  
 private int age;  
  
}

* **@Data :** A shortcut for @Getter, @Setter, @ToString, @EqualsAndHashCode, and @RequiredArgsConstructor (except for final fields only). It’s most commonly used to make a POJO clean and concise.
* **@NoArgsConstructor**: Automatically generates a public no-argument constructor (public Student() {}), required by frameworks like Spring and Jackson.
* **@AllArgsConstructor**: Generates a constructor with all fields as parameters. E.g., public Student(Long id, String name, String email, int age).

4. Creating Repository class, can later be replaced with jpa.

**@Repository**: Indicates this class deals with data (like a DAO). Spring detects it and makes it a bean.

package com.example.studentManagement.repository;  
  
import com.example.studentManagement.model.Student;  
import org.springframework.stereotype.Repository;  
  
import java.util.\*;  
  
@Repository // Marks class as a Spring-managed data component  
public class StudentRepository {  
  
 //like a temp db  
 private final Map<Long, Student> studentDb = new HashMap<>();  
 private Long currentId = 3L;  
  
 //adding some sample data  
 {  
 studentDb.put(1L, new Student(1L, "Palash","palash@gmail.com",32));  
 studentDb.put(2L, new Student(2L, "Raj","raj@gmail.com",23));  
 }  
  
 public List<Student> findAll() {  
 return new ArrayList<>(studentDb.values());  
 }  
  
 public Optional<Student> findById(Long id) {  
 return Optional.*ofNullable*(studentDb.get(id));  
 }  
  
 public Student save(Student student) {  
 if (student.getId() == null) {  
 student.setId(currentId++);  
 }  
 studentDb.put(student.getId(), student);  
 return student;  
 }  
  
 public void deleteById(Long id) {  
 studentDb.remove(id);  
 }  
  
 public boolean existsById(Long id) {  
 return studentDb.containsKey(id);  
 }  
}

5. Creating Service layer, Here we will have all logics of how to delete, update, fetch etc.

* @Service: Marks this class as a business service.
* @Autowired: Lets Spring inject dependencies automatically (in this case, the repository).

package com.example.studentManagement.service;  
  
import com.example.studentManagement.model.Student;  
import com.example.studentManagement.repository.StudentRepository;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
  
import java.util.List;  
import java.util.Optional;  
  
@Service // Business logic layer component  
public class StudentService {  
  
 @Autowired // Injects StudentRepository bean  
 private StudentRepository repository;  
  
 //to get all students  
 public List<Student> getAllStudents() {  
 return repository.findAll();  
 }  
  
 public Student getStudentById(Long id) {  
 return repository.findById(id).orElse(null);  
 }  
  
 public Student createStudent(Student student) {  
 return repository.save(student);  
 }  
  
 public Student updateStudent(Long id, Student updatedStudent) {  
 if (repository.existsById(id)) {  
 updatedStudent.setId(id);  
 return repository.save(updatedStudent);  
 }  
 return null;  
 }  
  
 public boolean deleteStudent(Long id) {  
 if (repository.existsById(id)) {  
 repository.deleteById(id);  
 return true;  
 }  
 return false;  
 }  
}

6. Creating Controller layer, Here we will have all APIs used by our system

* @RestController: Marks this class as a Controller

package com.example.studentManagement.controller;  
import com.example.studentManagement.model.Student;  
import com.example.studentManagement.service.StudentService;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.\*;  
import java.util.List;  
  
@RestController // @Controller + @ResponseBody  
@RequestMapping("/students") // Base path for all endpoints  
public class StudentController {  
  
 @Autowired  
 private StudentService service;  
  
 @GetMapping  
 public List<Student> getAllStudents() {  
 return service.getAllStudents();  
 }  
  
 @GetMapping("/{id}")  
 public Student getStudentById(@PathVariable Long id) {  
 return service.getStudentById(id);  
 }  
  
 @PostMapping  
 public Student createStudent(@RequestBody Student student) {  
 return service.createStudent(student);  
 }  
  
 @PutMapping("/{id}") //This is for update  
 public Student updateStudent(@PathVariable Long id, @RequestBody Student student) {  
 return service.updateStudent(id, student);  
 }  
  
 @DeleteMapping("/{id}") //This is for delete  
 public String deleteStudent(@PathVariable Long id) {  
 return service.deleteStudent(id) ? "Deleted" : "Student not found";  
 }  
}

* @**RestController**: Tells Spring this class handles REST API and returns data (not views). Combines @Controller + @ResponseBody for REST API responses
* @**RequestMapping**("/api/students"): Sets base URL for all methods.
* @**GetMapping**, @**PostMapping**, @**PutMapping**, @**DeleteMapping**: Maps HTTP methods to handler methods.
* @**PathVariable**: Extracts {id} from the URL.
* @**RequestBody**: Converts incoming JSON into a Student object automatically.

**Rest APIs**

| **HTTP Method** | **URL** | **Description** |
| --- | --- | --- |
| GET | /students | Get list of all students. |
| GET | /students/{id} | Get a student by their ID. |
| POST | /students | Create a new student. |
| PUT | /students/{id} | Update student details by ID. |
| DELETE | /students/{id} | Delete a student by ID. |

## 🔹 **Swagger Documentation**

Swagger/OpenAPI is a REST API documentation tool that auto-generates UI from code annotations.

**Benefits:**

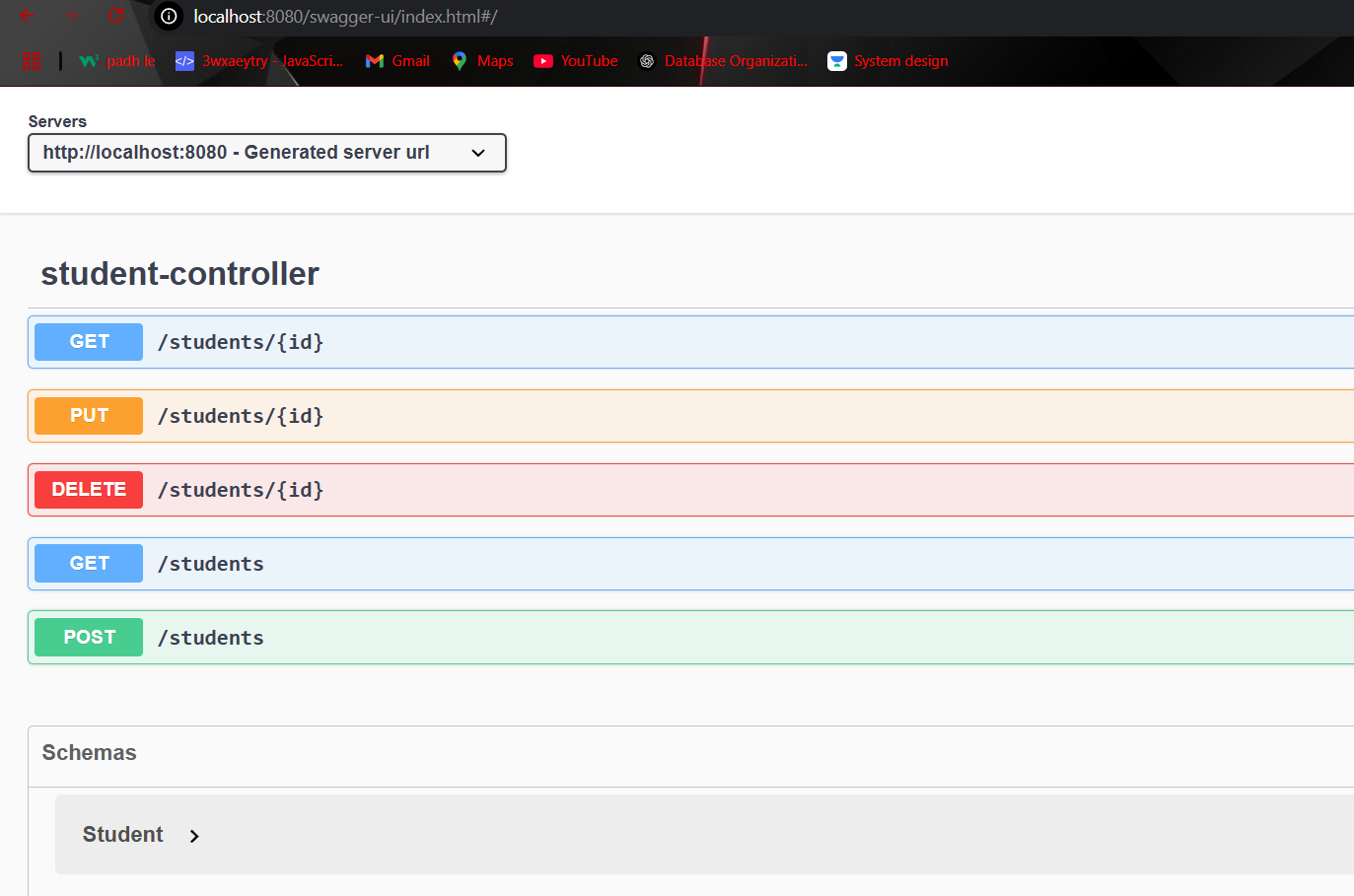
* Developer-friendly interface to test APIs
* Self-documenting code
* Client SDK generation

Depencency:

<dependency>  
 <groupId>org.springdoc</groupId>  
 <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>  
 <version>2.2.0</version> <!-- or latest -->  
</dependency>

Go to link: <http://localhost:8080/swagger-ui.html>

And we get



## 🔹 **Spring Boot Actuators**

Spring Boot Actuator is a sub-project of Spring Boot that provides production-ready features for monitoring and managing your application. It exposes built-in endpoints over HTTP or JMX to give insights into app’s internal state - like health, metrics, environment, and more.

**Benefits:**

* Monitor app health and status
* Track metrics (e.g., memory, CPU, request count)
* View system environment properties
* Manage app beans, mappings, logs, etc.
* Integrates well with monitoring tools like Prometheus, Grafana, New Relic

**Depencency:**

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-actuator</artifactId>  
</dependency>

Also in application.properties add

#For actuator  
management.endpoints.web.exposure.include=\*  
management.endpoint.shutdown.enabled=true

**Connections:**

| **URL** | **Purpose** |
| --- | --- |
| http://localhost:8080/actuator | List of all available actuator endpoints |
| http://localhost:8080/actuator/health | App health status (UP or DOWN) |
| http://localhost:8080/actuator/info | Custom app info from properties |
| http://localhost:8080/actuator/metrics | Exposes metrics like memory, threads |
| http://localhost:8080/actuator/env | Environment variables |
| http://localhost:8080/actuator/mappings | All controller request mappings |
| http://localhost:8080/actuator/beans | All loaded Spring beans |
| http://localhost:8080/actuator/loggers | Logging levels and controls |

**Use cases:**

If your app crashes, hangs, or misbehaves in production:

* Use /actuator/health to check DB/connection issues
* Use /actuator/metrics to see memory/CPU problems
* Use /actuator/env to view loaded environment profiles
* Use /actuator/loggers to increase debug log level at runtime

# **4. Spring JPA**

*Spring Data JPA is the easiest way to persist and query data using standard Java classes without writing SQL.*

Spring Data JPA is a Spring project that simplifies the use of JPA (Java Persistence API) for object-relational mapping (ORM). It removes boilerplate code and provides powerful abstractions to interact with databases using repositories.

**Key Concepts:**

| **Term** | **Description** |
| --- | --- |
| **Entity** | A class mapped to a database table using annotations |
| **Repository** | An interface to abstract data access and query operations |
| **Service** | A layer between controllers and repositories for business logic |
| **Transaction** | A unit of work that must succeed or fail as a whole |

🔹 **Entity**

An Entity is a Java class mapped to a database table using @Entity.

**Important Annotations**

|  |  |
| --- | --- |
| @Entity | Declares a class as a JPA entity, if table don’t exist it can create a table in db |
| @Id | Specifies the primary key |
| @GeneratedValue | Auto-generates primary key values |
| @Column | Maps a field to a table column |
| @Table | (Optional) Specifies custom table name |

@Entity  
@Table(name = "users")  
public class User {  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 private Long id;  
  
 @Column(nullable = false)  
 private String name;  
  
 private String email;  
}

**Repository Layer**

Repositories extend JpaRepository or CrudRepository to provide CRUD and pagination features.

| **Interface** | **Description** |
| --- | --- |
| CrudRepository<T, ID> | Basic CRUD (save, findById, delete) |
| JpaRepository<T, ID> | Extends CrudRepository with more features like pagination, flush, batch |

public interface UserRepository extends JpaRepository<User, Long> {  
 List<User> findByName(String name);  
}

Here in JpaRepository<T,ID> , replace T with Entity name and ID is data type of primary Key of that Entity.

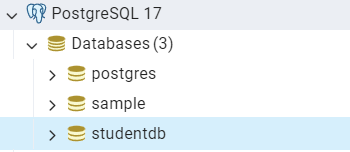
***CODE EXAMPLE:***

We will update Student Management System with JPA code.

1. Add maven dependency for jpa and postgresSql driver

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-jpa</artifactId>  
</dependency>  
<dependency>  
 <groupId>org.postgresql</groupId>  
 <artifactId>postgresql</artifactId>  
 <version>42.7.2</version>  
</dependency>

2. Setup data base (studentdb) from pgadmin



Add setup in application.properties.

* spring.jpa.hibernate.ddl-auto: allows to perform ddl, like if Entity table is not present, create a table in db.
* spring.jpa.show-sql=true: shows sql executed in console.

# PostgreSQL Configuration  
spring.datasource.url=jdbc:postgresql://localhost:5432/studentdb  
spring.datasource.username=postgres  
spring.datasource.password=1234  
  
# Hibernate Configuration  
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect  
spring.jpa.hibernate.ddl-auto=update   
spring.jpa.show-sql=true

3. Mark Student.java as Entity

package com.example.studentManagement.model;  
import jakarta.persistence.\*;  
import java.lang.\*;  
  
@Entity  
public class Student {  
  
 @Id //Mark id as primary key  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private Long id;  
 private String name;  
 private String email;  
 private int age;

public Student(){} // Required by JPA

public Student(Long id, String name, String email, int age) {  
 this.id = id;  
 this.name = name;  
 this.email = email;  
 this.age = age;  
 }  
  
 public Long getId() {  
 return id;  
 }  
 public void setId(Long id) {  
 this.id = id;  
 }  
  
 public int getAge() {  
 return age;  
 }  
 public void setAge(int age) {  
 this.age = age;  
 }  
  
 public String getEmail() {  
 return email;  
 }  
 public void setEmail(String email) {  
 this.email = email;  
 }  
  
 public String getName() {  
 return name;  
 }  
 public void setName(String name) {  
 this.name = name;  
 }  
}

JPA (Java Persistence API) uses reflection to instantiate entity objects, and it requires a no-argument constructor to create instances when loading data from the database.

4. Mark StudentRepository.java and extend JpaRepository

package com.example.studentManagement.repository;  
  
import com.example.studentManagement.model.Student;  
import org.springframework.stereotype.Repository;  
import org.springframework.data.jpa.repository.JpaRepository;  
  
@Repository  
public interface StudentRepository extends JpaRepository<Student, Long> {  
}

JpaRepository is a Spring Data interface that extends CrudRepository and PagingAndSortingRepository to provide a rich set of methods for working with JPA (Java Persistence API) entities — without writing any boilerplate code.

| **Feature** | **Description** |
| --- | --- |
| 🔁 **CRUD Operations** | Built-in methods like save(), findById(), findAll(), delete() |
| 🔍 **Query Derivation** | Auto-generates queries from method names like findByName() |
| 🔢 **Pagination & Sorting** | Easily paginate and sort using Pageable, Sort |
| 💬 **Custom Queries** | Write JPQL or native SQL using @Query |
| 📦 **Batch Operations** | saveAll(), deleteAllInBatch(), flush() for efficiency |
| 📌 **Transactional Support** | Works seamlessly with Spring’s @Transactional |
| 🧠 **Intelligent Proxy Generation** | Spring auto-generates implementation using **proxies at runtime** |

**Common build in methods for JpaRepository**

| **Method** | **Description** |
| --- | --- |
| save(entity) | Save or update an entity |
| findById(id) | Get one entity by primary key |
| findAll() | Get all entities |
| deleteById(id) | Delete entity by ID |
| count() | Count total records |
| existsById(id) | Check if entity exists |
| findAll(Pageable) | Paginate results |
| findAll(Sort) | Sort results |

5. Calling the repo methods via service

@Service  
public class StudentService {  
 @Autowired  
 StudentRepository repo;  
  
 public List<Student> getStudents(){  
 return repo.findAll();  
 }  
  
 public Student getStudentById(Long id){  
 return repo.findById(id).orElseThrow(() -> new RuntimeException("Student not found"));  
 }  
  
 public Student addStudent(Student student){  
 return repo.save(student);  
 }  
  
 public void removeStudent(Long id){  
 repo.deleteById(id);  
 }  
  
 public void updateStudentById(Long id, Student student) {  
 Student existing = getStudentById(id);  
 existing.setName(student.getName());  
 existing.setEmail(student.getEmail());  
 existing.setAge(student.getAge());  
 repo.save(existing);  
 }  
}

6. Controller class will need no change, just few changes in some return type

@RestController  
@RequestMapping("/students")  
public class StudentController {  
  
 @Autowired StudentService service;  
  
 @GetMapping  
 public List<Student> getAllStudents(){  
 return service.getStudents();  
 }  
  
 @GetMapping("/{id}")  
 public Student getAllStudents(@PathVariable Long id){  
 return service.getStudentById(id);  
 }  
  
 @PostMapping  
 public Student addUser(@RequestBody Student student){  
 return service.addStudent(student);  
 }  
  
 @DeleteMapping("/{id}")  
 public void deleteStudent(@PathVariable Long id){  
 service.removeStudent(id);  
 }  
  
 @PutMapping("/{id}")  
 public void updateStudent(@PathVariable Long id, @RequestBody Student student){  
 service.updateStudentById(id,student);  
 }  
}

## 🔹 **JPQL(Java Persistence Query Language)**

**JPQL (Java Persistence Query Language)** is a platform-independent object-oriented query language defined as part of JPA (Java Persistence API). Unlike SQL which queries tables and columns, **JPQL queries entities and their properties**.

* SQL: operates on *tables* and *columns*
* JPQL: operates on *Java classes* (entities) and fields

**Syntax**

SELECT s FROM Student s WHERE s.name = :name

* Student s: refers to the Entity class
* s.name: is the field in the entity (not a DB column)
* :name: is a named parameter

**Eg:** Lets say , till now we were searching by id, but wants to search by name. So we can write query for same using JPQL. Remember, repo is interface here

@Repository  
public interface StudentRepository extends JpaRepository<Student, Long> {  
 // JPQL Query  
 @Query("SELECT s FROM Student s WHERE s.name = :name")  
 List<Student> findByName(String name);  
}

* @Query: Defines a custom JPQL query

| **Aspect** | **JPQL** | **Native SQL** |
| --- | --- | --- |
| Query Type | Entity-based | Table/Column-based |
| Portability | Highly portable across DBs | Tied to specific SQL dialects |
| Use-case | Common queries, simple joins | Complex joins, DB functions |
| Annotation | @Query("JPQL here") | @Query(value="SQL", nativeQuery=true) |

## 🔹 **Derived Query Methods**

Spring Data JPA can **automatically** **implement queries** for you by analyzing the method name in the repository interface. These methods are called query derivation methods.

Eg: in place of writing this:

@Query("SELECT s FROM Student s WHERE s.name = :name")  
List<Student> findByName(String name);

We can simple write:

List<Student> findByName(String name);

You don’t need to write @Query unless you need custom logic, spring Data JPA will automatically generate the correct JPQL at runtime: *SELECT s FROM Student s WHERE s.name = ?1*

**How does Spring do this?**

Spring parses the method name using the pattern: findBy<FieldName>. Then it maps it to the corresponding entity field. For example:

* findById(Long id) → SELECT s FROM Student s WHERE s.id = ?1
* findByName(String name) → SELECT s FROM Student s WHERE s.name = ?1
* findByAgeGreaterThan(int age) → SELECT s FROM Student s WHERE s.age > ?1

**Common Keywords used**

| **Keyword** | **Description** |
| --- | --- |
| And, Or | Combine conditions |
| Is, Equals | Equality |
| Between | Range queries |
| LessThan | < comparison |
| GreaterThan | > comparison |
| Like, Containing | Pattern matching (LIKE in SQL) |
| OrderBy | Sorting results |
| Not, IsNot | Negation |

List<Student> findByName(String name);  
List<Student> findByNameAndAge(String name, int age);  
List<Student> findByNameContaining(String keyword);  
List<Student> findByAgeGreaterThanEqual(int age);

# **5. Spring Configuration and profiling**

## 🔹 **application.properties and application.yml**

Spring Boot uses application.properties or application.yml files in the src/main/resources directory to centralize configuration.

You can define:

* Server properties (port, context path)
* Database connection details
* Logging levels
* Profiles
* Custom key-value configs

Key differences:

| **application.properties** | **application.yml** |
| --- | --- |
| Key-Value format | YAML hierarchy format |
| Easier for scripts | More readable for nested configs |
| key=value | key: value |

Example:-

1. application.properties

server.port=8081  
server.servlet.context-path=/api  
spring.application.name=MyApp  
logging.level.org.springframework=DEBUG

2. application.yml

server:  
 port: 8081  
 servlet:  
 context-path: /api  
spring:  
 application:  
 name: MyApp  
logging:  
 level:  
 org.springframework: DEBUG

**Notes:**

* Default file is application.properties.
* You can use **both .properties and .yml** but not for the same keys (avoid confusion).
* Use .yml for hierarchical data (e.g., database, logging).
* Spring Boot will load application.properties or application.yml automatically from the classpath.

**Important properties**

1. You can set the **port** and **context path** of your application easily using:

server.port=8085  
server.servlet.context-path=/service

This makes your app accessible at: http://localhost:8085/service

## 🔹 **External Configuration**

Spring Boot supports **externalizing configuration**, which means values can come from:

* application.properties or application.yml
* **Command-Line Arguments**
* **Environment Variables**
* @Value or @ConfigurationProperties

This allows **dynamic configuration** for environments like dev, test, or prod.

**Command Line Example:**

java -jar demo.jar --server.port=9095 --custom.message="Deployed in QA"

**Environment Variable Example (Linux/macOS):**

export SERVER\_PORT=7070  
export SPRING\_PROFILES\_ACTIVE=prod

Spring will automatically map SERVER\_PORT to server.port.

**Notes:**

* Command-line args override all.
* Environment variables are useful in containers (Docker/Kubernetes).
* Useful in CI/CD pipelines for dynamic deployments.

Priority order of configuration: (Highest to Lowest Priority):

1. Command-line arguments
2. Environment variables
3. application-<profile>.properties
4. application.properties
5. Default values in code

## 🔹 **Profiles**

We use **profiling** in Spring Boot to define and manage **multiple configurations** (e.g., dev, test, prod) for different environments, allowing the application to load specific beans or settings based on the active profile.

**Profiles help you separate environment-specific logic**, like using an in-memory database in development and a real database in production.

**Uses Cases of profiling:**

* Use **different database configurations** for dev and prod
* Use **mock beans** for testing and real beans for production
* Enable or disable **debugging tools** only in dev profile
* Load different **application properties** per environment

**How to create different profiles:**

Create 2 different application.properties, each for dev and prod

1. application-dev.properties:

server.port=8080  
spring.datasource.url=jdbc:mysql://localhost:3306/devdb  
spring.datasource.username=devuser  
spring.datasource.password=devpass

2. application-prod.properties:

server.port=80  
spring.datasource.url=jdbc:mysql://prod-db:3306/proddb  
spring.datasource.username=produser  
spring.datasource.password=prodpass

**Activate required profile:**

1. set in default application.properties

spring.profiles.active=dev

2. via command line

java -jar demo.jar --spring.profiles.active=prod

**@Profile**

Creating beans based on profile. The annotation @Profile("dev") tells Spring to **only load that bean or configuration class** when the **dev profile is active**.

@Configuration  
@Profile("dev")  
public class DevConfig {  
 @Bean  
 public DataSource devDataSource() {  
 return new H2DataSource(); // In-memory DB for dev  
 }  
}

This bean will **only be created** if: spring.profiles.active=dev

You can create multiple profile or even negate profile:

@Profile({"dev", "test"})  
  
@Profile("!prod") // Loaded if profile is NOT "prod"

# **6. Spring AOP**

*AOP (Aspect-Oriented Programming) is a* ***programming paradigm*** *in Spring used to* ***modularize cross-cutting concerns*** *— like logging, security, transactions, exception handling - that are scattered across many classes (called cross-cutting concerns).*

* Spring AOP allows you to separate such concerns from business logic using aspects, making code cleaner and more maintainable.
* Spring AOP allows you to inject additional behaviors before, after, or around method execution in Spring-managed beans — without modifying the actual method code.

**Use Cases of AOP:**

| **Use Case** | **Why AOP Is Ideal** |
| --- | --- |
| **Logging** | Keep logs cleanly separate from business logic |
| **Performance Metrics** | Measure method execution time |
| **Security** | Check user permissions before execution |
| **Transactions** | Add @Transactional declaratively |
| **Retry Mechanisms** | Wrap methods to retry on failure |
| **Audit Trails** | Track changes to data or method calls |

## **🔹 Core AOP Concepts**

|  |  |
| --- | --- |
| **Aspect** | A module that encapsulates behaviors affecting multiple classes (a concern). |
| **Join Point** | A point during execution — like method call or exception — where an aspect can be applied. |
| **Advice** | The action taken by an aspect at a join point. Example: @Before, @After, @Around. |
| **Pointcut** | A predicate that matches join points. Used to define where advice should be applied. |
| **Weaving** | Linking aspects with other application types or objects at runtime or compile time. |
| **Proxy** | Spring AOP uses proxies to implement AOP. The original bean is wrapped in a proxy object. |

***CODE EXAMPLE:***

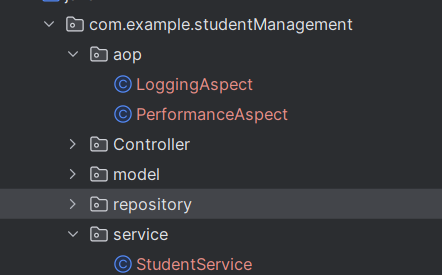
We will add AOP to our Student Management Application

1. Add Dependency

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-aop</artifactId>  
</dependency>

2. Create an aspect for logging, here we will target service classes.

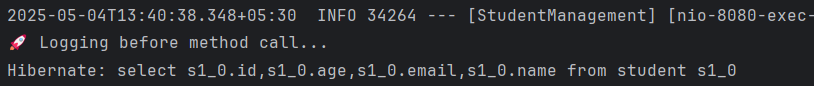
Created 2 aspect files, for now will try LoggingAspect



Below code will run before any method in service class runs.

@Aspect  
@Component  
public class LoggingAspect {  
  
 @Before("execution(\* com.example.studentManagement.service.\*.\*(..))")  
 public void logBeforeMethodCall() {  
 System.*out*.println("🚀 Logging before method call...");  
 }  
}

Output:



**@Before("execution(\* com.example.service.\*.\*(..))")**

This is a **pointcut expression** saying:

* Match all methods
* In all classes under com.example.service
* With any arguments

**Different Pointcut Expression:**

| **Expression** | **Meaning** |
| --- | --- |
| execution(\* com.example..\*.\*(..)) | Match any method in any class in package com.example or its subpackages |
| execution(public \* \*(..)) | Match any public method |
| execution(\* \*(..)) | Match any method |
| execution(\* \*..\*Service.\*(..)) | Match all methods in classes ending with Service |

**Different Advices:**

| **Advice Type** | **Annotation** | **Purpose** |
| --- | --- | --- |
| **Before** | @Before | Run code before method |
| **After** | @After | Run code after method |
| **AfterReturning** | @AfterReturning | Run code after successful method execution |
| **AfterThrowing** | @AfterThrowing | Run code when method throws exception |
| **Around** | @Around | Run code before and after method, even control execution |

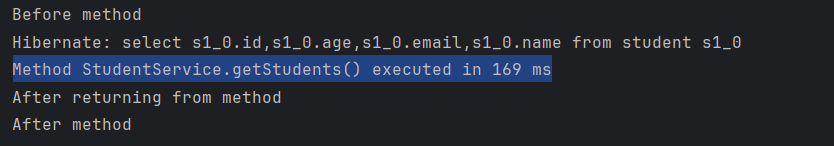
**Using different aspects:**

@Aspect  
@Component  
public class LoggingAspect {  
  
 @Before("execution(\* com.example.studentManagement.service.StudentService.\*(..))")  
 public void beforeAdvice() {  
 System.*out*.println("Before method");  
 }  
  
 @After("execution(\* com.example.studentManagement.service.StudentService.\*(..))")  
 public void afterAdvice() {  
 System.*out*.println("After method");  
 }  
  
 @AfterReturning("execution(\* com.example.studentManagement.service.StudentService.\*(..))")  
 public void afterReturningAdvice() {  
 System.*out*.println("After returning from method");  
 }  
  
 @AfterThrowing("execution(\* com.example.studentManagement.service.StudentService.\*(..))")  
 public void afterThrowingAdvice() {  
 System.*out*.println("After throwing exception");  
 }  
  
}

3. Creating an aspect for testing performance

@Aspect  
@Component  
public class PerformanceAspect {  
  
 @Around("execution(\* com.example.studentManagement.service.StudentService.\*(..))")  
 public Object logExecutionTime(ProceedingJoinPoint joinPoint) throws Throwable {  
 long startTime = System.*currentTimeMillis*();  
  
 Object result = joinPoint.proceed(); // Proceed to actual method  
  
 long endTime = System.*currentTimeMillis*();  
 String methodName = joinPoint.getSignature().toShortString();  
  
 System.*out*.println("Method " + methodName + " executed in " + (endTime - startTime) + " ms");  
  
 return result;  
 }  
}

we get this:



Explanation:

| **Element** | **Meaning** |
| --- | --- |
| @Aspect | Declares this class as an Aspect |
| @Component | Registers it as a Spring-managed bean |
| @Around(...) | Advice that wraps method execution |
| execution(\* ... ) | Pointcut expression — matches **all methods** in StudentService |
| ProceedingJoinPoint | Used in @Around advice to control method execution flow |
| joinPoint.proceed() | Calls the actual method; you can modify inputs/outputs if needed |
| joinPoint.getSignature() | Gives method name for logging |

**Limitations of Spring AOP:**

* Only works with Spring-managed beans and not with other java objects.
* Only public and protected methods are intercepted, can’t work on private methods
* Only method-level interception is supported, don’t work on field level

## **🔹 SLF4J**

SLF4J stands for **Simple Logging Facade for Java**.

It is not a logging framework itself, but a facade or abstraction for various logging frameworks such as:

* Log4j
* Logback
* java.util.logging (JUL)
* TinyLog, etc.

**Uses:**

* **Decouples** logging implementation from application code.
* You can switch between logging frameworks **without changing your code**.
* Promotes consistency and compatibility in large Java applications.

**SLF4J setup:**

<!-- SLF4J API -->  
<dependency>  
 <groupId>org.slf4j</groupId>  
 <artifactId>slf4j-api</artifactId>  
</dependency>

import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
  
@SpringBootApplication   
public class StudentManagementApplication {  
 private static final Logger *logger* = LoggerFactory.*getLogger*(StudentManagementApplication.class);  
   
 public static void main(String[] args) {  
 *logger*.info("Application started"); //using slf4j for logging  
 SpringApplication.*run*(StudentManagementApplication.class, args);  
 }  
}

**Why SlF4j is better than system.out.println**

| **Feature** | **System.out.println** | **SLF4J with Logback** |
| --- | --- | --- |
| Logging Levels | ❌ Not available | ✅ TRACE, DEBUG, INFO, WARN, ERROR |
| Performance | ❌ Poor | ✅ Efficient (string formatting is lazy) |
| File/Appender Output | ❌ Console only | ✅ Files, databases, monitoring systems |
| Configurability | ❌ Hard-coded | ✅ Configurable via application.properties |
| Production Usage | ❌ Bad Practice | ✅ Industry Standard |