Ashwini Upadhyay

Spring core
Spring boot
Spring AOP & MVC
Spring Security
Spring Security with JWT
Docker
Microservices
Kafka with Docker and Microservices

Course Content

1. Spring Core
2. Spring Boot
3. Spring Data JPA
4. Spring Web MVC
5. RESTful Services
6. Microservices
7. Spring Security
8. Spring Cloud
9. Cache
10. Cloud deployment
11. Docker
12. Kafka
13. Kubernetes (K8)
14. Unit testing

Spring Core

Lec 1

Project : Business logic + Common logic

Business Logic:

- Refers to the core functionality specific to the application's domain (e.g., user authentication, order processing, payment validation).

Common Logic:

- Reusable functionality that supports the application but isn't domain-specific (e.g., logging, validation, authentication).

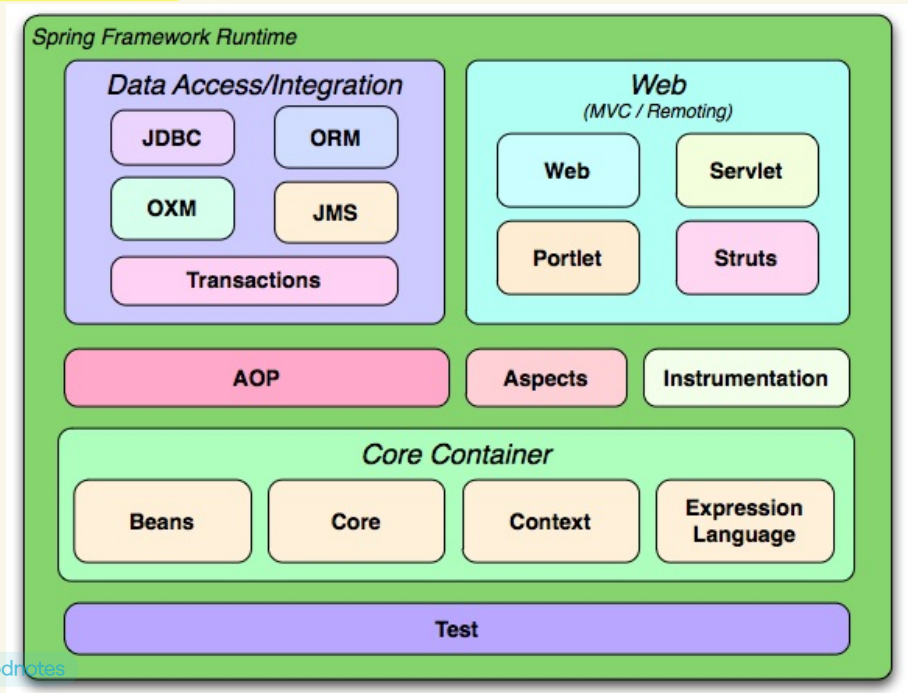
What is framework

A **framework** is a pre-built set of tools, libraries, and best practices that help developers build software applications faster and more efficiently by providing structure and reusable components.

Real-world example:
Think of a **house framework** where the foundation, walls, and roof are already built, and you just need to design the rooms and decorate the house. Similarly, in programming, a framework provides the basic structure of the application, and developers fill in the specific details.

Example in programming:

- **Spring Framework**: A Java-based framework that helps build enterprise-level applications by providing tools for things like dependency injection, security, and database management.



Here are one-liner definitions for each tool, along with real-world examples:

1. **Maven**:

A build automation tool for Java projects that manages dependencies, builds, and packaging.
Example: It simplifies compiling and packaging Java applications, like building a `.jar` file for deployment.

2. **Gradle**:

A modern build automation tool, similar to Maven, but with more flexibility and support for other languages like Groovy and Kotlin.
Example: Used in Android development to manage dependencies and compile code into APK files.

3. **Jira**:

A project management and issue tracking tool used to plan, track, and manage software development tasks and bugs.
Example: Teams use Jira to assign tasks, track progress, and resolve bugs in a collaborative software project.

4. **Jenkins**:

An open-source automation server used to build, test, and deploy software through continuous integration and continuous delivery (CI/CD).
Example: Automatically running tests and deploying code every time a developer pushes changes to a repository.

5. **JMeter**:

A performance testing tool used for load testing and measuring the performance of web applications.
Example: It simulates multiple users accessing a website to see how well it handles traffic under load.

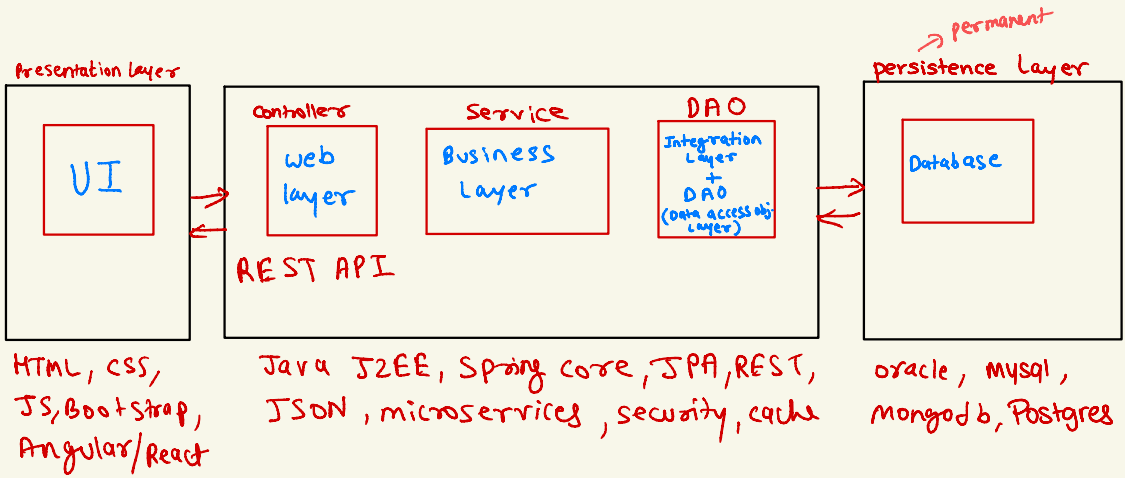
6. **Postman**:

A tool for testing and interacting with APIs, allowing users to send HTTP requests and analyze responses.
Example: Developers use Postman to test API endpoints to ensure they return correct data before integration.

7. **SonarQube**:

A tool for continuous code quality inspection, providing feedback on code quality, security vulnerabilities, and technical debt.
Example: It runs static code analysis on a codebase to identify potential issues and areas for improvement before deployment.

Application Architechture



Definition which describes a layered architecture:

1. Presentation Layer:

The UI layer that handles user interactions and displays information.
Technologies: HTML, CSS, JS, Bootstrap, Angular/React.

2. Web Layer:

Handles HTTP requests and provides REST APIs for communication between the client and server.
Technologies: Java J2EE, Spring Core, REST, JSON, Microservices, Security.

3. Business Layer:

The layer responsible for implementing business logic and processing data.
Technologies: Java business logic, microservices, security.

4. Integration Layer + DAO:

Manages the communication between the business logic and the database, including data access and integration with other services.
Technologies: Data Access Objects (DAO), integration tools.

5. Persistence Layer:

The database layer that stores and retrieves application data in a permanent storage solution.
Technologies: Oracle, MySQL, MongoDB, Postgres.

Architecture

1. Monolithic Architecture:

A traditional software architecture where the entire application is built as a single, unified unit.
Example: A banking application where user interface, business logic, and database access are all part of one large application, often hard to scale or modify.

2. Microservices Architecture:

A modern software architecture where an application is divided into smaller, independent services that communicate over a network.
Example: An e-commerce platform with separate microservices for user authentication, product management, order processing, and payment, each developed, deployed, and scaled independently.

Spring modules:

13. Test: Unit Testing framework

Core Modules

1. Spring Core: Provides the fundamental features of the framework, including Dependency Injection (DI) and Inversion of Control (IoC), Bean life cycle, Bean Scope, Auto-wiring.
2. Spring Context: Offers enterprise-level support like internationalization, event handling, and resource management.

Deal with configurations

Data Access Modules

3. Spring JDBC: Simplifies database access by providing templates for common JDBC tasks.
4. Spring ORM: Integrates ORM frameworks (Hibernate, JPA, MyBatis) for object-relational mapping.

AOP (Aspect-Oriented Programming)

5. Spring AOP: Supports aspect-oriented programming for cross-cutting concerns like logging, transactions, and security.

Web Modules

6. Spring Web MVC: Implements the Model-View-Controller (MVC) pattern for building robust web applications.
7. Spring REST: Provides support for building RESTful web services using annotations like @RestController.

Security

→ login (OAuth2.0)

→ roles (admin)

8. Spring Security: Handles authentication, authorization, and securing applications against vulnerabilities.

JWT

Batch Processing

9. Spring Batch: Provides batch processing capabilities like task scheduling, chunk-based processing, and retry mechanisms. (notifications)

Data Access Layer

10. Spring Data JPA: Simplifies JPA-based data access by providing repository interfaces and query abstraction.

Social Integration

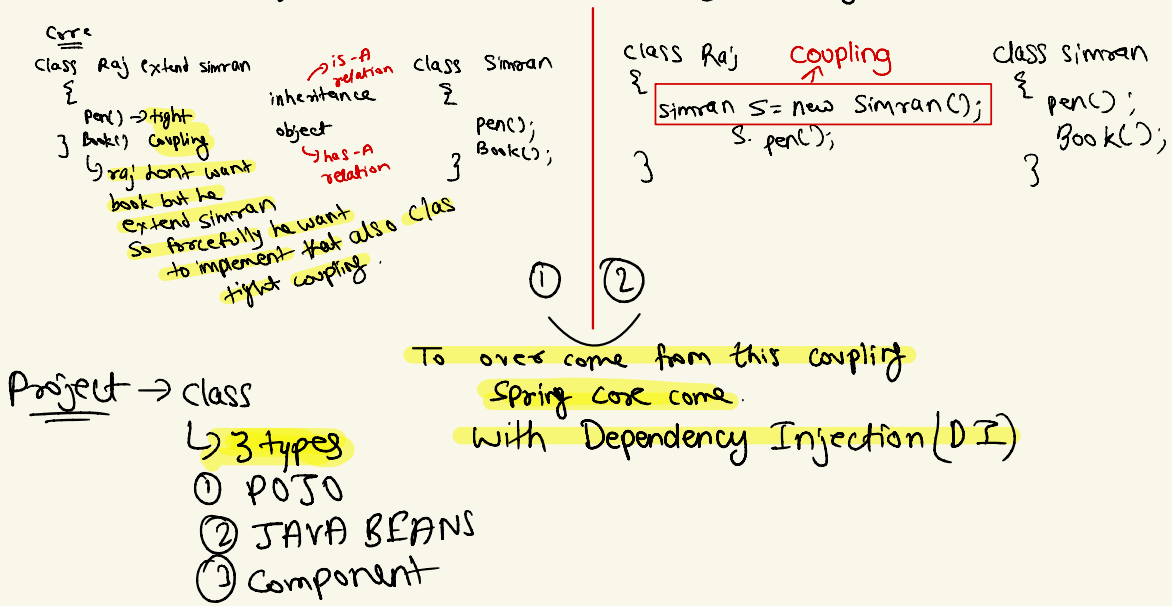
11. Spring Social: Integrates social platforms like Facebook, Twitter, and LinkedIn into applications.

Cloud Computing

12. Spring Cloud: Supports microservices development with tools for service discovery, configuration management, circuit breakers, and more. discovery service, Gateway, Load Balancers.

Spring Core : It is all about managing dependencies among the classes with loosely coupling

Lec 2



Project → class

↳ 3 types

1. POJO
2. JAVA BEANS
3. Component

POJO → Plain Old Java Object

1. **Definition**: Any simple Java class that can be compiled using JDK.
2. **Explanation**: A POJO refers to a Java object that does not depend on any framework, library, or special annotation. Unlike Java classes such as Servlets or EJBs (which require special APIs), POJOs are simple objects adhering to standard Java conventions.

Java Beans

1) Any Java class which follow specific rules
   a) Class must implement the `java.io.Serializable` interface
   b) class must have private variables
   c) private variable should have public getter and setter
   d) class should have **No-Argument Constructor**

Components

↳ contains business logic

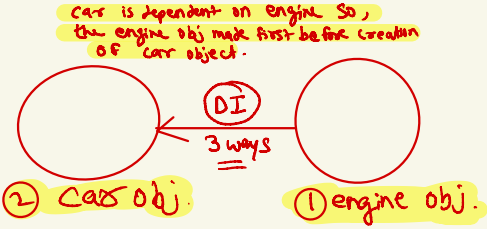
Ex: Controller, Service, DAO classes.

- Controller → class deal with Req & Res
- Service → Business Logic
- DAO → DB Logic

Dependency Injection

↳ It is the process of injecting one class object into another class is called DI

DI
1. ↳ Setter DI
2. ↳ Constructor DI
3. ↳ Field DI



eg

```
public class Car {
    private Engine eng;
    public void setEng(Engine eng){
        this.eng=eng;
    }
    public void drive(){
        int start = eng.start();
        if(start > 1){
            System.out.println("Car");
        }else{
            System.out.println("Engine not started");
        }
    }
}
```

① eng object → reference

} setter DI

```
public class main{
    psvm(String[] args)
    {
        Car car = new Car();
        car.setEng(new Engine());
        car.drive();
    }
}
```

eg

```
public class Car {
    private Engine eng;
    public Car(Engine eng){
        this.eng=eng;
    }
    public void drive(){
        int start = eng.start();
        if(start > 1){
            System.out.println("Car");
        }else{
            System.out.println("Engine not started");
        }
    }
}
```

② } constructor DI

```
public class main {
    psvm (String [] args) {
        Car car = new Car (new Engine());
        car.drive();
    }
}
```

eg

```
public class Car {
    @Autowire
    private Engine eng;

    public void drive(){
        int start = eng.start();
        if(start > 1){
            System.out.println("Car");
        }else{
            System.out.println("Engine not started");
        }
    }
}
```

③

we have to implement

```
public class main {
    psvm (String [] args) {
        Car car = new Car ();
        car.drive();
    }
}
```

this internal work of Spring

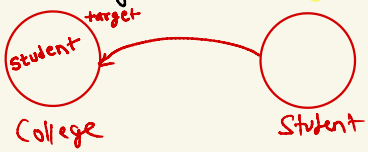
IoC container

Lec 3

↳ IoC → Inversion of control.

↳ Why → responsible for DI in spring Application.

DI → create object and inject into the target bean class.

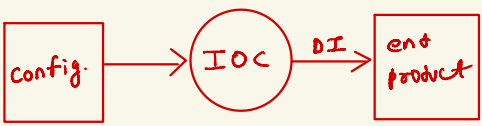


Note: IOC will manage life cycle of spring Bean.

Note: We need to provide Java classes + Bean config. as input for IOC then IOC perform DI and provide spring Bean which are ready to use.

Type of Ioc

1. Bean factory (outdated)
2. ApplicationContext (recommended)



How to create Spring Core App

1) Create a maven project
2) Add Spring context dependency
3) Create Required class
4) Create Beans.xml
5) Create main (driver) to start IOC container.

Key Points

1. **Dependencies**: Ensure Spring Core dependency is added.
2. **Bean Definition**: Define beans in XML or Java-based configuration.
3. **IoC Container**: Use `ApplicationContext` to manage beans.
4. **Business Logic**: Implement it in bean classes.

1. `Engine` Interface

The interface defines the behavior that both engines will implement.

java Copy code

```java
// Step 1: Define the Engine interface
public interface Engine {
    // Method to be implemented by engine types
    void start();
}
```

2. `DieselEngine` Class

Implements the `Engine` interface for diesel engine functionality.

java Copy code

```java
// Step 2: Create DieselEngine class implementing Engine
public class DieselEngine implements Engine {
    @Override
    public void start() {
        // Step 6: Print message when the diesel engine starts
        System.out.println("Diesel engine starting...");
    }
}
```

3. `PetrolEngine` Class

Implements the `Engine` interface for petrol engine functionality.

java Copy code

```java
// Step 3: Create PetrolEngine class implementing Engine
public class PetrolEngine implements Engine {
    @Override
    public void start() {
        // Step 6: Print message when the petrol engine starts
        System.out.println("Petrol engine starting...");
    }
}
```

4. `Car` Class

Represents the car that depends on the `Engine` interface.

java Copy code

```java
// Step 4: Create Car class with dependency on Engine
public class Car {
    private Engine engine;

    // Default constructor
    public Car() {
        System.out.println("Car object created."); // Step 5: Print when the Car object is i
    }

    // Setter for Dependency Injection
    public void setEngine(Engine engine) {
        this.engine = engine; // Step 7: Inject the engine dependency into the car
    }

    // Method to start the car
    public void startCar() {
        System.out.println("Car is starting..."); // Step 8: Print when the car starts
        engine.start(); // Step 9: Call the start method of the injected engine
    }
}
```

5. `beans.xml` File

The configuration file defines the beans and injects the dependencies.

xml Copy code

```xml
<?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">

    <!-- Step 10: Define DieselEngine bean -->
    <bean id="dieselEngine" class="com.example.DieselEngine" />

    <!-- Step 10: Define PetrolEngine bean -->
    <bean id="petrolEngine" class="com.example.PetrolEngine" />

    <!-- Step 11: Define Car bean and inject dieselEngine -->
    <bean id="car" class="com.example.Car">
        <property name="engine" ref="dieselEngine" /> <!-- Change to petrolEngine for petrol
    </bean>

</beans>
```

6. `App` Class

The entry point to load the Spring IoC container and run the application.

java Copy code

```java
// Step 12: Create App class to load Spring configuration and start the application
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App {
    public static void main(String[] args) {
        // Step 13: Load Spring configuration file
        ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");

        // Step 14: Get the Car bean from the Spring IoC container
        Car car = (Car) context.getBean("car");

        // Step 15: Start the car (calls the engine's start method)
        car.startCar();
    }
}
```

Sample Output

If `dieselEngine` is injected, the output will be:

csharp Copy code

```
Car object created.
Car is starting...
Diesel engine starting...
```

If you change the `ref` to `petrolEngine`, the output will be:

csharp Copy code

```
Car object created.
Car is starting...
Petrol engine starting...
```

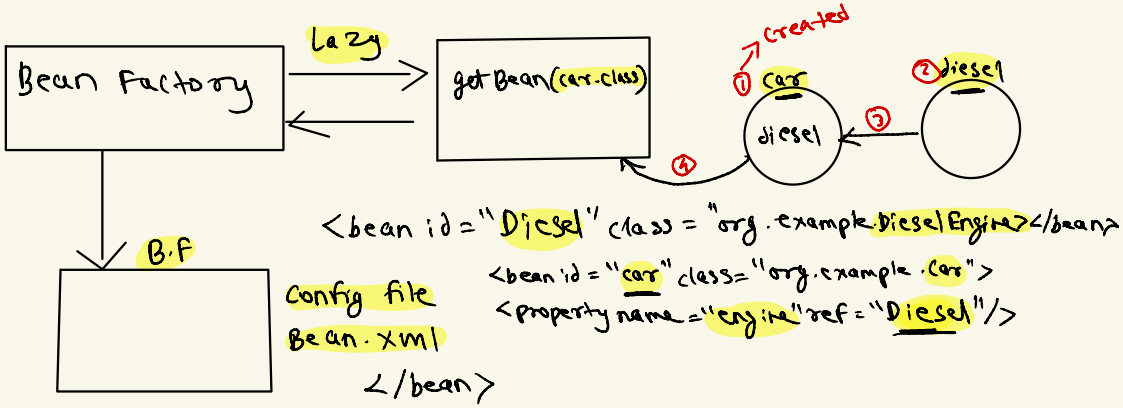
Simple Spring Core First App

Lec 4

Beanfactory (outdated)
↳ IOC container
↳ follow lazy loading concept that means when we request then only it will create Bean object.

Application Context (recommended)
↳ IOC container
↳ follow Eager loading

Polymorphism
- overloading [compile time] [eager] [early binding]
- overriding [runtime] [lazy] [late binding]




```
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;

public class BeanFactoryClassExample {
    public static void main(String[] args) {
        // Load Spring configuration
        BeanFactory factory = new XmlBeanFactory(new ClassPathResource("beans.xml"));

        // Retrieve a bean
        Object bean = factory.getBean("myBean");

        // Using java.lang.Class to inspect the bean
        Class<?> beanClass = bean.getClass();

        // Print the class name
        System.out.println("Bean Class Name: " + beanClass.getName());

        // Check if the bean is of a specific type
        if (MyBean.class.isAssignableFrom(beanClass)) {
            System.out.println("The bean is an instance of MyBean.");
        }
    }
}

class MyBean {
    public void doWork() {
        System.out.println("MyBean is working!");
    }
}
```

Key Points in the Example

1. **Class Inspection**:
   - The `bean.getClass()` call retrieves the runtime `Class` object of the instantiated bean.
2. **Type Checking**:
   - `isAssignableFrom()` checks if the bean's class is compatible with the specified type.
3. **Dynamic Nature**:
   - The actual runtime type of the bean can differ if Spring applies proxies (e.g., for AOP), but the `Class` object allows you to analyze its structure dynamically.

```java
public class ClassExample {
    public static void main(String[] args) {
        // Create an instance of a class
        Person person = new Person("John", 25);

        // Get the Class object using the getClass() method
        Class<?> personClass = person.getClass();

        // Print the class name
        System.out.println("Class Name: " + personClass.getName());

        // Print the simple name of the class
        System.out.println("Simple Name: " + personClass.getSimpleName());

        // Print the package name
        System.out.println("Package Name: " + personClass.getPackageName());

        // List declared fields
        System.out.println("Declared Fields:");
        Arrays.stream(personClass.getDeclaredFields())
              .forEach(field -> System.out.println("- " + field.getName()));

        // List declared methods
        System.out.println("Declared Methods:");
        Arrays.stream(personClass.getDeclaredMethods())
              .forEach(method -> System.out.println("- " + method.getName()));
    }
}
```

```java
class Person {
    private String name;
    private int age;

    // Constructor
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Getter for name
    public String getName() {
        return name;
    }

    // Getter for age
    public int getAge() {
        return age;
    }
}
```

```
Class Name: ClassExample$Person
Simple Name: Person
Package Name: (Default Package)
Declared Fields:
- name
- age
Declared Methods:
- getName
- getAge
```

1. **Class Object**:
   - The `getClass()` method on an object returns the `Class` object representing its runtime type.
2. **Retrieving Class Details**:
   - `getName()`: Returns the fully qualified name of the class (e.g., `ClassExample$Person` in this case because `Person` is an inner class).
   - `getSimpleName()`: Returns the class name without the package (e.g., `Person`).
   - `getPackageName()`: Returns the package name.
3. **Fields and Methods**:
   - `getDeclaredFields()`: Returns all declared fields (including private fields) in the class.
   - `getDeclaredMethods()`: Returns all declared methods in the class.

Java.lang.Class

Simple example demonstrating the use of java.lang.Class to obtain information about a class at runtime.

1. Spring Beans:

Objects managed by the Spring IoC container, representing the core of an application.

2. IoC Container:

A framework responsible for managing the lifecycle and dependencies of Spring beans through Inversion of Control.

3. Dependency Injection (DI) with Types:

The process of injecting dependencies into beans, supported by three types: Constructor Injection, Setter Injection, and Field Injection.

4. How to Start IoC:

Use an implementation of ApplicationContext or BeanFactory to load the bean configuration and manage beans.

5. Bean Configuration File:

An XML or Java-based file (e.g., beans.xml or annotated class) defining bean metadata and dependencies for the IoC container.

resources/Beans.xml

```
<bean id="engine" class="org.example.PetrolEngine"></bean>

<bean id="carobject" class="org.example.Car" scope="prototype" autowire="byName">
    <property name="engine" ref="petrol" />
    <constructor-arg name="engine" ref="Diesel" />
</bean>
```

field DI (autowire)

→ Setter DI

→ Constructor DI

Note: When we use Dependency Injection & Constructor Injection both at same time then Setter Injection will override Constructor Injection.

Bean Scope

1. Singleton (default)
2. prototype
3. request
4. session

Bean Scope

1. Singleton (default): A single shared instance of the bean is created and reused across the application context.
2. Prototype: A new instance of the bean is created every time it is requested.
3. Request: A new bean instance is created for each HTTP request (web applications).
4. Session: A single bean instance is maintained per HTTP session (web applications). New object for every session.

Autowiring DI (recommended)

1) inject dependent bean into target in 2 ways
   * Manual wiring → Setter DI (<property>), constructor DI (<constructor>) → bean.xml ref = ...
   * Autowiring
2) Autowiring means IOC container will identify dependent bean and it will inject into target bean.
   (we no need to use ref attribute in bean.xml config file)
3) Autowiring will work based on
   1. byName
   2. byType
   3. constructor
   4. No.

Autowiring → DI → IOC Container ← BF (BeanFactory), AC (ApplicationContext) ● use this now-a-days)

Spring → container → BF (Bean factory) / AC (Application Context) — use any one

Lec 5

(BF, AF)

config.xml → parse → Token

Java class

Spring Beans

AC → xml file (eager loading)

BF → req → id = petrol

- only Petrol obj created
- associated with petrol obj created

lifecycle

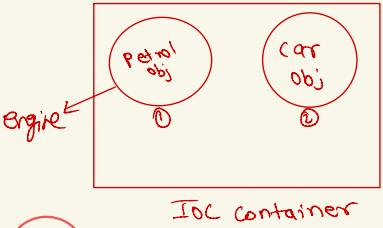
DI object inject

* manual → SI, CI
* Field injection → Autowiring
  - byname
  - by Type
  - by constructor

Spring core terminology → IOC

```
① <bean id="engine" class="org.example.PetrolEngine"></bean>
② <bean id="carobject" class="org.example.Car" scope="prototype" autowire="byName">
```

Application Context Read in this direction

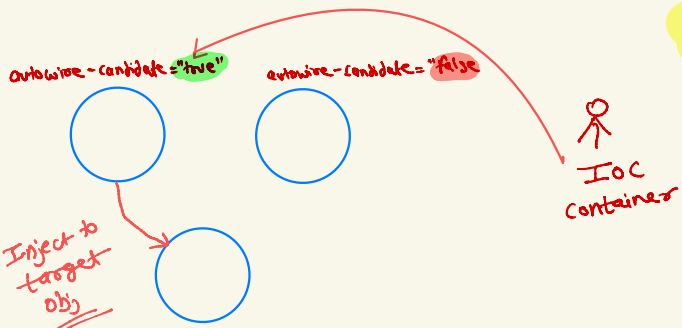


Java class (Petrol) ⇒ id = petrol → car

id = Petrol

IOC container

Spring → terminology



Field DI

autowire

- byname
- byType
- byconstructor
- byno

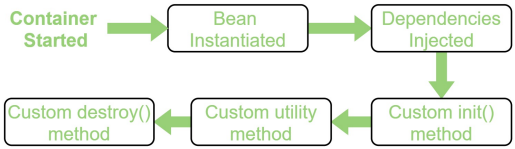
Spring Bean Life Cycle

↳ manage by IOC

→ init()

→ destroy

1. xml approach
2. programatic approach
3. Annotation approach



The **lifecycle of a Spring bean** is managed by the IoC container, and it follows these steps:

1. **Bean Instantiation**:
   - The IoC container creates an instance of the bean using the configured constructor or factory method.
2. **Dependency Injection**:
   - The container injects dependencies into the bean's properties based on the configuration (e.g., via setter methods or constructor arguments).
3. **Bean Initialization**:
   - If the bean implements the `InitializingBean` interface, the `afterPropertiesSet()` method is called.
   - If an `init-method` is specified in the bean configuration, it is invoked.
4. **Bean Usage**:
   - The bean is ready to be used by the application.
5. **Bean Destruction**:
   - When the container is shutting down, it calls the `destroy()` method (if the bean implements `DisposableBean`) or any custom `destroy-method` specified in the configuration.

Example of Lifecycle Configuration in XML

xml Copy

```xml
<bean id="myBean" class="com.example.MyBean"
      init-method="init" destroy-method="cleanup">
</bean>
```

Using Annotations

- `@PostConstruct`: Marks the initialization method.
- `@PreDestroy`: Marks the destruction method.

Example:

java Copy

```java
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

public class MyBean {
    @PostConstruct
    public void init() {
        System.out.println("Bean is initialized");
    }

    @PreDestroy
    public void cleanup() {
        System.out.println("Bean is destroyed");
    }
}
```

Summary

1. **Instantiation → Dependency Injection → Initialization → Usage → Destruction.**

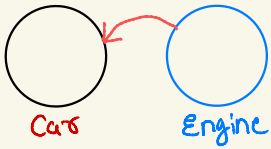
Summary

1. **Instantiation → Dependency Injection → Initialization → Usage → Destruction.**
2. Initialization and destruction hooks can be defined via interfaces, configuration, or annotations.

Spring Dependency Injection

Lec 6

→ Setter DI



IoC

BeanFactory
↳ will create a bean obj. when the getBean() method is called make it lazy initialization

Application Context
↳ load all Beans and create obj. at the Startup that's why it is fast and recommended

Note: Constructor DI make Engine obj. first the inject to Car obj.

attribute
* Prototype → in(Beans.xml) → scope = "Prototype"
↳ on Req → new obj. created

IoC container
* BeanFactory → all bean lazy load.
↳ compulsory to write getBean in main method to load Beans. (in classpath lazy load)

Note: But in ApplicationContext IOC loads all the beans and creates object at the time of startup. (Eager Loading)

Spring → xml (hard way to develop App)

Spring → Annotations (easy way)
↳ release in 1.5 v
↳ xml alternative

1) @Configuration → To represent class as Configuration class

2) @ComponentScan → To identify Spring Bean class in project based "Base package". (com.example)
Java class → scan → (classpath: Application of local JVM's)

3) @Component → put label to tell Spring this is component.

Here's a real-world analogy to understand these annotations:

1. `@Configuration`:

Think of it as the blueprint or the *recipe book* where all the instructions (bean definitions) for building objects (beans) are written.

2. `@ComponentScan`:

Like a *search operation*, where you tell Spring to look for specific items (components) in a specific area (package) of your warehouse.

3. `@Component`:

It's like putting a *label* on an item in your warehouse, saying, "This is a usable item" so it can be easily identified and used when needed.

Spring Annotation

Lec 7

1) @Configuration
2) @ComponentScan
3) @Component
4) @Service
5) @Repository
6) @Scope
7) @Autowired
8) @Qualifier
9) @Controller / @RestController
10) @Primary
11) @Bean
12) @Entity
13) @Table
14) @Id
15) @Column

Spring Boot = Spring framework (core) - xml config.

↳ Approach to develop Spring Based App with less config.

↳ Spring Boot is not replacement for Spring framework. Spring Boot develop on top of Spring framework (core)

↳ Note: Spring Core can be used in Spring Boot

↳ Spring Boot Advantages

↳ ① Less config. & no xml config.

↳ ② Embedded Server

↳ ③ Starter dependency — JPA Starter, web-Starter, spring-Starter

↳ ④ Auto config. (no xml file)

↳ ⑤ Actuator (production ready feature)

@SpringBootApplication

- Definition: It is a meta-annotation in Spring Boot that combines three essential annotations:

1. @SpringBootConfiguration: Equivalent to @Configuration, it indicates that the class provides Spring configuration.
2. @EnableAutoConfiguration: Enables Spring Boot's auto-configuration feature, which automatically configures beans based on the classpath dependencies.
3. @ComponentScan: Scans the package and its sub-packages for Spring components (e.g., @Component, @Service, @Repository, @Controller).

```
package com.spark.firstapp;

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

@SpringBootApplication
public class FirstAppApplication {
    public static void main(String[] args) {
        // Start the Spring Boot application
        SpringApplication.run(FirstAppApplication.class, args);
    }
}
```

Lec 8

Code Explanation

1. @SpringBootApplication
   - Marks the class as the main entry point for a Spring Boot application.
   - Combines @SpringBootConfiguration, @EnableAutoConfiguration, and @ComponentScan.
2. SpringApplication.run()
   - Bootstraps the Spring Boot application.
   - Responsibilities:
     - Displays the Spring Boot banner during startup.
     - Starts the Spring IoC container (ApplicationContext).
     - Returns a reference to the IoC container (context).

Spring Boot Starters

Spring Boot starters are pre-configured dependencies that simplify application setup:

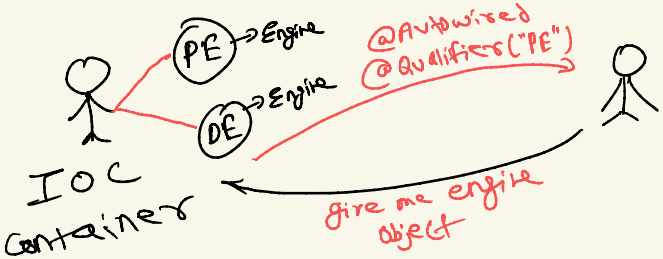
1. spring-boot-starter
   - Provides basic dependencies for building standalone Spring Boot applications.
2. spring-web
   - Adds support for building traditional web applications using Spring MVC.
3. webflux
   - Adds support for building reactive applications using Project Reactor.

Key Takeaways

- Main Method: Starts the Spring Boot application using SpringApplication.run().
- Starters: Simplify project setup by bundling commonly used libraries (e.g., for web apps or reactive programming).
- IoC Container: Manages bean creation and dependency injection.

Spring Boot

Lec 1 - 2

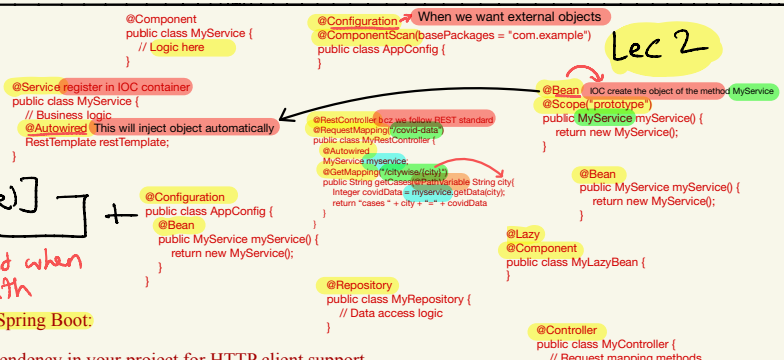


* spring-boot-devtools
  ↳ use to rerun app automatically
  like nodemon in nodejs

1. `@PostConstruct`: Marks a method to be executed *after* the bean is initialized (once dependencies are injected). run after constructor but before app run.
2. `@PreDestroy`: Marks a method to be executed *before* the bean is destroyed (for cleanup tasks). run before object deletion.
3. `@Component`: Marks a class as a Spring-managed bean, eligible for dependency injection.
4. `@Autowired`: Automatically injects a bean (by type) into the dependent class. → object in term of java. → bean in term of spring.
5. `@Qualifier`: Specifies the exact bean to inject when multiple candidates of the same type exist.

Lec 2

@ Component
@ Service
@ RestController
@ Beans [method level] + @ Configuration
↳ to reg. in IOC container and when we want Bean in classpath

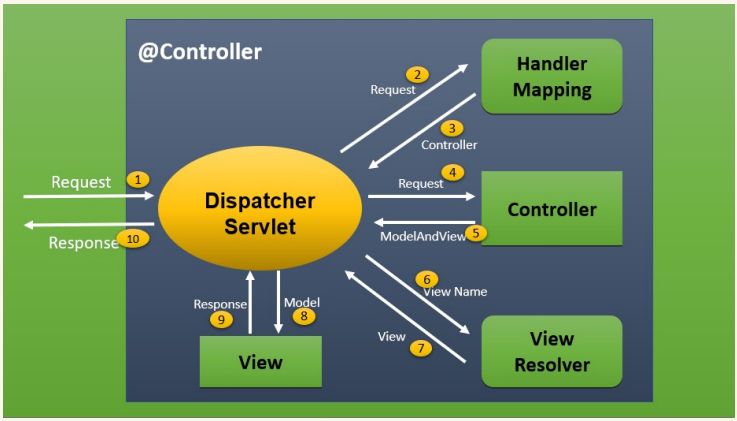


Here's a theoretical guide to consuming an API in Spring Boot:

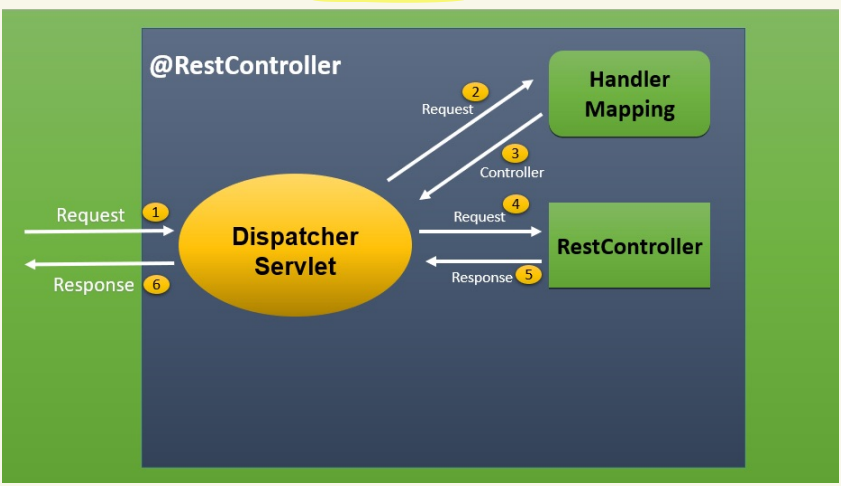
1. Add Dependency: Include the Spring Web dependency in your project for HTTP client support.
2. Choose HTTP Client:
   - Use RestTemplate for synchronous/blocking API calls (simpler for traditional use cases).
   - Use WebClient for asynchronous/non-blocking API calls (recommended for reactive programming).
3. Configure Client:
   - Define a RestTemplate or WebClient bean in a configuration class to use throughout the application.
4. Make API Call:
   - Use RestTemplate methods like getForObject, postForObject, or exchange.
   - Use WebClient methods like get(), post(), and retrieve() with fluent chaining.
5. Handle Response: Parse and handle the API response (e.g., as a String, JSON, or mapped to a custom Java object).
6. Exception Handling: Add proper error handling using try-catch for RestTemplate or .onErrorResume for WebClient.
7. Inject Client in Services: Inject and use the configured RestTemplate or WebClient in your services or components.
8. Test the Endpoint: Test your code by exposing it via a Spring Controller or directly from the service.

Lec 3

Old MVC



New MVC



Spring MVC Request Flow:

1. **DispatcherServlet**: The central Servlet that handles all incoming HTTP requests.
2. **Handler Mapping**: Determines which controller should handle the request.
3. **Controller** (or **@RestController**): Processes the request and prepares the response.
4. **Response**: Returned to the client via DispatcherServlet.

1. **@RestController**: Combines `@Controller` and `@ResponseBody`, indicating that the class handles HTTP requests and directly returns the response as JSON or XML.
2. **@RequestMapping**: Maps HTTP requests to handler methods in the controller.
3. **@GetMapping / @PostMapping / @PutMapping / @DeleteMapping**: Specialized annotations for HTTP methods (GET, POST, PUT, DELETE) to map specific request paths to controller methods.
4. **@ResponseBody**: Indicates that the return value of a method should be serialized and sent directly in the HTTP response body.

Here's a list of the annotations used in the code along with their main purposes:

1. Entity Layer

- @Entity: Marks a class as a JPA entity (maps to a database table).
- @Id: Denotes the primary key of the entity.
- @GeneratedValue(strategy = GenerationType.IDENTITY): Configures auto-generation of primary key values.
- @Column(nullable = false): Specifies column properties, e.g., non-null.

2. Repository Layer

- @Repository (implicitly provided by extending JpaRepository): Identifies a class as a repository layer component for database operations.

3. Service Layer

- @Service: Marks a class as a service layer component for business logic.
- @Autowired: Injects dependencies automatically.

4. Controller Layer

- @RestController: Combines @Controller and @ResponseBody to create RESTful web services.
- @RequestMapping("/api/todos"): Specifies the base URL for all endpoints in the controller.
- @GetMapping: Handles HTTP GET requests.
- @PostMapping: Handles HTTP POST requests.
- @PutMapping: Handles HTTP PUT requests.
- @DeleteMapping: Handles HTTP DELETE requests.
- @PathVariable: Binds a method parameter to a path variable in the URL.
- @RequestBody: Maps HTTP request body to a Java object.

5. Lombok

- @Data: Generates getters, setters, toString, equals, and hashCode methods.
- @NoArgsConstructor: Generates a no-argument constructor.
- @AllArgsConstructor: Generates a constructor with all fields as arguments.

6. Configuration Layer

- @Configuration: Marks a class as a source of Spring configuration.
- @Bean: Defines a method that returns a Spring-managed bean.

These annotations collectively simplify configuration, dependency injection, and development of RESTful APIs in Spring Boot.

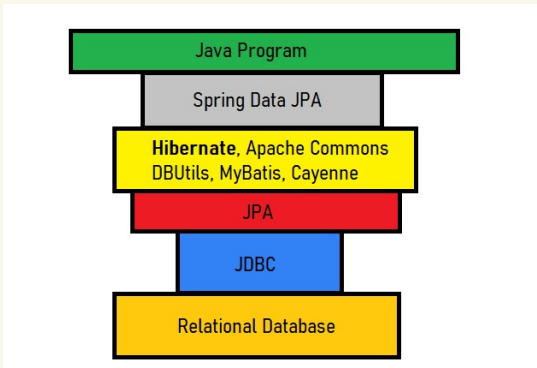
Lec 4

What is Spring Data JPA?

Spring Data JPA is a part of the larger Spring Data family, designed to simplify working with databases in Java. It abstracts common data access logic and provides a powerful repository-based programming model.

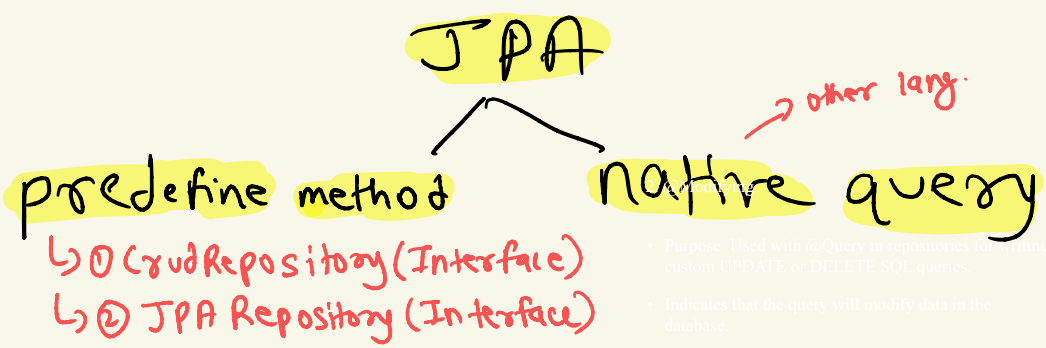
Key Features of Spring Data JPA

1. Simplifies database operations with minimal boilerplate code.
2. Provides a **Repository** interface to handle CRUD operations.
3. Supports JPQL (Java Persistence Query Language) and native SQL queries.
4. Supports pagination and sorting.
5. Enables custom query methods using method names (query derivation).



How It Works

1. **Entity Class (`User`)**: Defines the database table structure.
2. **Repository (`UserRepository`)**: Provides CRUD and custom query methods.
3. **Service Layer**: Handles business logic.
4. **Controller Layer**: Handles HTTP requests and responses.
5. **Spring Data JPA**: Automatically generates SQL queries based on the repository interface.



- Purpose: Used with @Query in repositories for defining custom UPDATE or DELETE SQL queries.
- Indicates that the query will modify data in the database.

1.@Transactional:

- Ensures a series of operations are part of a single transaction.
- Rolls back changes on runtime exceptions.

2.@Modifying:

- Required for UPDATE, INSERT, or DELETE queries in Spring Data JPA.
- Must be combined with @Transactional to ensure database integrity.

Entity & Table

1. @Entity: Marks a class as a JPA entity (maps to a database table).
2. @Table(name = "table_name"): Specifies the table name for the entity (optional if the class name matches the table name).

Primary Key & ID Generation

3. @Id: Specifies the primary key field.
4. @GeneratedValue(strategy = GenerationType.*): Defines how the primary key is generated (AUTO, IDENTITY, SEQUENCE, TABLE).
5. @SequenceGenerator: Configures a database sequence for primary key generation.
6. @TableGenerator: Configures a table for primary key value generation.

Column Mapping

7. @Column(name = "column_name"): Maps a field to a database column (optional).
8. @Transient: Marks a field as non-persistent (ignored by JPA).
9. @Lob: Maps a field to a large object, e.g., BLOB or CLOB.

Relationships

10. @OneToOne: Defines a one-to-one relationship between entities.
11. @OneToMany: Defines a one-to-many relationship (a collection of related entities).
12. @ManyToOne: Defines a many-to-one relationship (a single related entity).
13. @ManyToMany: Defines a many-to-many relationship between entities.
14. @JoinColumn(name = "column_name"): Specifies the foreign key column for a relationship.
15. @JoinTable: Defines a join table for many-to-many relationships.

Constraints

16. @NotNull: Ensures a column cannot be null.
17. @UniqueConstraint: Enforces uniqueness on a column or combination of columns.
18. @Embedded: Embeds an object within the entity.

Inheritance

19. @Inheritance(strategy = InheritanceType.*): Specifies inheritance mapping strategy (SINGLE_TABLE, JOINED, TABLE_PER_CLASS).
20. @DiscriminatorColumn: Defines a column to store the type of subclass in a single-table inheritance strategy.
21. @MappedSuperclass: Marks a class as a parent entity that other entities inherit (not a table itself).

Lifecycle

22. @PrePersist: Method annotated runs before the entity is persisted.
23. @PostPersist: Method annotated runs after the entity is persisted.
24. @PreUpdate: Method annotated runs before the entity is updated.
25. @PostUpdate: Method annotated runs after the entity is updated.
26. @PreRemove: Method annotated runs before the entity is removed.
27. @PostRemove: Method annotated runs after the entity is removed.
28. @PostLoad: Method annotated runs after the entity is loaded from the database.

Caching

29. @Cacheable: Marks an entity as cacheable for second-level cache.

Versioning

30. @Version: Enables optimistic locking by adding a version column.

Create Table in MySQL

```
CREATE TABLE users (
  id BIGINT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(50) NOT NULL,
  email VARCHAR(255) NOT NULL UNIQUE
);
```

Entity Class

```
package com.example.demo.entity;

import jakarta.persistence.*;

@Entity // Specifies that this is a database entity
@Table(name = "users") // Maps this class to the "users" table
public class User {

    @Id // Primary key
    @GeneratedValue(strategy = GenerationType.IDENTITY) // Auto-increment
    private Long id;  or int id;

    @Column(nullable = false, length = 50) // Maps to a column in the table
    private String name;

    @Column(unique = true, nullable = false) // Unique constraint for email
    private String email;

    // 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;
    }
}
```

Repository

```
package com.example.demo.repository;

import com.example.demo.entity.User;
import org.springframework.data.jpa.repository.JpaRepository;

public interface UserRepository extends JpaRepository<User, Long> {
    // Custom query method to find a user by email
    User findByEmail(String email);
}
```

Integer id

Note:

```
ArrayList<Integer> al = new ArrayList<>();
al.add(10);
al.get(10);
```

al.add(10); int → auto Boxing (Wrapper class) → Integer

Integer → unBoxing → int

int → auto Boxing → Integer

Integer int → unBoxing al.get(0); → Index

Data type

Service Layer

```
package com.example.demo.service;

import com.example.demo.entity.User;
import com.example.demo.repository.UserRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class UserService {

    @Autowired
    private UserRepository userRepository;

    public List<User> getAllUsers() {
        return userRepository.findAll();
    }

    public User getUserById(Long id) {
        return userRepository.findById(id).orElse(null);
    }

    public User saveUser(User user) {
        return userRepository.save(user);
    }

    public void deleteUser(Long id) {
        userRepository.deleteById(id);
    }
}
```

Controller Layer

```
package com.example.demo.controller;

import com.example.demo.entity.User;
import com.example.demo.service.UserService;
import
org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/users")
public class UserController {

    @Autowired
    private UserService userService;

    @GetMapping // Get all users
    public List<User> getAllUsers() {
        return userService.getAllUsers();
    }

    @GetMapping("/{id}") // Get a user by ID
    public User getUserById
```

Lec 5

JPA CRUD

Create the Entity

```
package com.example.crudapp.entity;

import jakarta.persistence.*;

@Entity // Specifies this class as a database entity
public class Item {

    @Id // Primary key
    @GeneratedValue(strategy = GenerationType.IDENTITY) // Auto-generate ID
    private Long id;

    @Column(nullable = false) // Column cannot be null
    private String name;

    private String description;

    // 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 getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }
}
```

Create the Repository

```
package com.example.crudapp.repository;

import com.example.crudapp.entity.Item;
import org.springframework.data.jpa.repository.JpaRepository;

public interface ItemRepository extends JpaRepository<Item, Long> {
}
```

Configure Application Properties

```
# application.properties

# H2 database configuration
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

# Auto-create tables
spring.jpa.hibernate.ddl-auto=update

# H2 console
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
```

Maven Dependencies

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

Test

Test the Application

Run the application and test the following endpoints using Postman or cURL:

1. GET /api/items - Fetch all items.
2. GET /api/items/{id} - Fetch an item by ID.
3. POST /api/items - Create a new item.
4. PUT /api/items/{id} - Update an existing item.
5. DELETE /api/items/{id} - Delete an item.

Example body

```
{
    "name": "Example Item",
    "description": "An example description"
}
```

Create the Controller

```
package com.example.crudapp.controller;

import com.example.crudapp.entity.Item;
import com.example.crudapp.service.ItemService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/items")
public class ItemController {

    @Autowired
    private ItemService itemService;

    // Get all items
    @GetMapping
    public List<Item> getAllItems() {
        return itemService.getAllItems();
    }

    // Get item by ID
    @GetMapping("/{id}")
    public ResponseEntity<Item> getItemById(@PathVariable Long id) {
        Item item = itemService.getItemById(id);
        if (item != null) {
            return ResponseEntity.ok(item);
        }
        return ResponseEntity.notFound().build();
    }

    // Create a new item
    @PostMapping
    public Item createItem(@RequestBody Item item) {
        return itemService.saveItem(item);
    }

    // Update an item
    @PutMapping("/{id}")
    public ResponseEntity<Item> updateItem(@PathVariable Long id, @RequestBody Item updatedItem) {
        Item existingItem = itemService.getItemById(id);
        if (existingItem != null) {
            existingItem.setName(updatedItem.getName());
            existingItem.setDescription(updatedItem.getDescription());
            return ResponseEntity.ok(itemService.saveItem(existingItem));
        }
        return ResponseEntity.notFound().build();
    }

    // Delete an item
    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteItem(@PathVariable Long id) {
        if (itemService.getItemById(id) != null) {
            itemService.deleteItem(id);
            return ResponseEntity.noContent().build();
        }
        return ResponseEntity.notFound().build();
    }
}
```

Create the Service Layer

```
package com.example.crudapp.service;

import com.example.crudapp.entity.Item;
import com.example.crudapp.repository.ItemRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class ItemService {

    @Autowired
    private ItemRepository itemRepository;

    // Get all items
    public List<Item> getAllItems() {
        return itemRepository.findAll();
    }

    // Get item by ID
    public Item getItemById(Long id) {
        return itemRepository.findById(id).orElse(null);
    }

    // Create or update an item
    public Item saveItem(Item item) {
        return itemRepository.save(item);
    }

    // Delete an item
    public void deleteItem(Long id) {
        itemRepository.deleteById(id);
    }
}
```

Maven Dependencies

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
</dependency>
```

Create a Data Model

```
package com.example.apifetchapp.model;

public class Post {
    private int id;
    private int userId;
    private String title;
    private String body;

    // Getters and Setters
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public int getUserId() {
        return userId;
    }

    public void setUserId(int userId) {
        this.userId = userId;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getBody() {
        return body;
    }

    public void setBody(String body) {
        this.body = body;
    }
}
```

Fetch All Posts

URL: http://localhost:8080/api/posts

Fetch Post by ID

URL: http://localhost:8080/api/posts/1

External API Example

API URL: https://jsonplaceholder.typicode.com/posts

Create a Service to Fetch API Data

```
package com.example.apifetchapp.service;

import com.example.apifetchapp.model.Post;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

import java.util.Arrays;
import java.util.List;

@Service
public class ApiFetchService {

    private final RestTemplate restTemplate;

    public ApiFetchService() {
        this.restTemplate = new RestTemplate();
    }

    // Fetch all posts from the external API
    public List<Post> fetchPosts() {
        String apiUrl = "https://jsonplaceholder.typicode.com/posts";
        Post[] posts = restTemplate.getForObject(apiUrl, Post[].class);
        return Arrays.asList(posts); // Convert array to list
    }

    // Fetch a specific post by ID
    public Post fetchPostById(int id) {
        String apiUrl = "https://jsonplaceholder.typicode.com/posts/" + id;
        return restTemplate.getForObject(apiUrl, Post.class);
    }
}
```

Create a Controller

```
package com.example.apifetchapp.controller;

import com.example.apifetchapp.model.Post;
import com.example.apifetchapp.service.ApiFetchService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/posts")
public class ApiFetchController {

    @Autowired
    private ApiFetchService apiFetchService;

    // Endpoint to fetch all posts
    @GetMapping
    public List<Post> getAllPosts() {
        return apiFetchService.fetchPosts();
    }

    // Endpoint to fetch a post by ID
    @GetMapping("/{id}")
    public Post getPostById(@PathVariable int id) {
        return apiFetchService.fetchPostById(id);
    }
}
```

Key Concepts and Explanation

1. RestTemplate: A Spring class used for making RESTful API calls.
   - getForObject: Sends a GET request and maps the response to a Java object.
   - Post[].class: Maps a JSON array to a Java array.
2. Data Mapping: The Post class represents the JSON structure of the API response.
3. Controller-Service Pattern:
   - Controller: Handles HTTP requests and responses.
   - Service: Encapsulates business logic, like API calls.

ACID Properties in JPA (Short Notes)

1. **Atomicity**:
   - Ensures all operations in a transaction succeed or none are applied.
   - Managed using `@Transactional`.
   - Rollback happens automatically if an exception occurs.

Example:

```
@Transactional
public void performTransaction() {
    repository.save(entity1);
    repository.save(entity2);
    if (someError) throw new RuntimeException(); // Rolls back both saves
}
```

2. **Consistency**:
   - Maintains database integrity (constraints, relationships).
   - Enforced using annotations like `@NotNull`, `@UniqueConstraint`, and database rules.

Example:

```
@Column(nullable = false) // Ensures value cannot be null
private String name;
```

3. **Isolation**:
   - Prevents interference between concurrent transactions.
   - Controlled by database isolation levels (default is **Read Committed**).
   - Adjust with Spring's `@Transactional(isolation = Isolation.SERIALIZABLE)`.
4. **Durability**:
   - Changes made by a committed transaction are permanently saved.
   - Handled by the database, ensuring data is not lost even if the system crashes after commit.

SOLID Principles (Short Notes)

1. **Single Responsibility Principle (SRP)**
   - **Definition**: A class should have only one reason to change.
   - **Explanation**: Each class should handle only one responsibility.
   - **Example**: Separate classes for `InvoiceCalculator` and `InvoicePrinter`.

2. **Open/Closed Principle (OCP)**
   - **Definition**: A class should be open for extension but closed for modification.
   - **Explanation**: Add new functionality via inheritance or composition without modifying existing code.
   - **Example**: Use interfaces or abstract classes for extension.

3. **Liskov Substitution Principle (LSP)**
   - **Definition**: Subtypes should be substitutable for their base types.
   - **Explanation**: Derived classes must not break the behavior of the parent class.
   - **Example**: If `Bird` has `fly()`, a subclass `Penguin` should not break this behavior.

4. **Interface Segregation Principle (ISP)**
   - **Definition**: A class should not be forced to implement methods it doesn't use.
   - **Explanation**: Break large interfaces into smaller, more specific ones.
   - **Example**: Split `Animal` interface into `Flyable` and `Swimmable`.

5. **Dependency Inversion Principle (DIP)**
   - **Definition**: High-level modules should not depend on low-level modules; both should depend on abstractions.
   - **Explanation**: Use interfaces to decouple dependencies.
   - **Example**: A `PaymentService` should depend on `PaymentProcessor` interface, not on specific implementations like `PayPalProcessor`.

These principles improve code maintainability, scalability, and flexibility.

Short Points with Real-Life Ideology in an E-commerce Example

Lec 6

controller/
Purpose: Manages incoming HTTP requests.
E-commerce Ideology: Serves as the storefront, where customers interact with product listings, shopping carts, and checkout processes via APIs.

service/
Purpose: Contains business logic.
E-commerce Ideology: Acts as the store manager, handling tasks like processing orders, managing inventory, and applying business rules.

repository/
Purpose: Interfaces for data access.
E-commerce Ideology: Functions as the database clerk, retrieving and storing product, order, and user data efficiently.

dao/
Purpose: Custom Data Access Objects.
E-commerce Ideology: Handles specialized data retrieval, like generating sales reports or fetching complex product recommendations.

dto/
Purpose: Data Transfer Objects.
E-commerce Ideology: Acts as the packaging for products, ensuring data sent to customers is secure, relevant, and properly formatted.

model/
Purpose: Entity classes representing data models.
E-commerce Ideology: Represents the inventory and customer records, mirroring the real-world items and users in the digital database.

config/
Purpose: Configuration classes.
E-commerce Ideology: Sets up the shop's environment, configuring security measures, payment gateways, and other essential settings.

utils/
Purpose: Utility classes and helper functions.
E-commerce Ideology: Provides tools like calculators and formatters, akin to cash registers and labeling machines in a physical store.

exception/
Purpose: Custom exception classes.
E-commerce Ideology: Manages unexpected situations gracefully, like handling out-of-stock items or failed payments, ensuring customer satisfaction.

interceptor/
Purpose: Request interceptors.
E-commerce Ideology: Acts as the security guard, checking credentials and permissions before allowing access to certain areas of the store.

filter/
Purpose: Servlet filters.
E-commerce Ideology: Functions like entrance protocols, ensuring only valid and safe requests enter the application.

aspect/
Purpose: AOP aspects.
E-commerce Ideology: Oversees cross-cutting concerns like security checks and logging, similar to surveillance systems monitoring the store.

listener/
Purpose: Event listeners.
E-commerce Ideology: Responds to events like new orders or inventory updates, triggering necessary actions like restocking or sending notifications.

validator/
Purpose: Validation logic.
E-commerce Ideology: Ensures all transactions and data entries are correct and comply with business rules, like verifying a coupon's validity.

constants/
Purpose: Application constants.
E-commerce Ideology: Stores fixed values like tax rates or standard shipping fees used throughout the store's operations.

enums/
Purpose: Enumerated types.
E-commerce Ideology: Defines fixed categories or statuses, such as payment methods or order states, ensuring consistency.

mapper/
Purpose: Entity-DTO mappers.
E-commerce Ideology: Translates internal data structures to customer-friendly formats, like wrapping a product in gift packaging.

annotation/
Purpose: Custom annotations.
E-commerce Ideology: Marks special sections or items with specific behaviors, like tagging premium products or sale items.

Lec 7

- **CORS** controls who can access your resources from different origins. It must be configured for browser-based applications accessing your API.

  Key Points:

  - Use `allowedOrigins("*")` cautiously, as it allows all origins.
  - Ensure proper configuration for security-sensitive applications.
  - Credentials (`allowCredentials(true)`) can only be sent if specific origins are allowed.

- **CSRF** protects against unauthorized actions by validating tokens. It's crucial for stateful applications but often disabled for REST APIs.

  Key Points:

  - CSRF protection is essential for stateful applications.
  - For stateless REST APIs (using JWT or OAuth), disabling CSRF is a common practice.
  - Use the CSRF token in headers or forms for secure requests.

Aspect	CORS	CSRF
Scope	Protects APIs from unauthorized cross-origin requests.	Protects authenticated users from forged requests.
Enforced By	Web browsers (via the `Origin` header).	Server-side application.
Purpose	Ensures legitimate origins access resources.	Ensures legitimate requests are made by authenticated users.

Lec 1

AOP

Aspect-Oriented Programming (AOP) in Spring Boot is a programming paradigm that allows you to separate cross-cutting concerns (like logging, security, or transaction management) from your core business logic. It helps keep your code clean and modular by enabling you to add these concerns dynamically without modifying the actual business logic.

Key Concepts of AOP:

1. Aspect: A module that contains cross-cutting concerns.
2. Advice: The action taken by an aspect at a particular join point.
   - Before: Runs before the method execution.
   - After: Runs after the method execution.
   - Around: Wraps the method execution and allows custom behavior before and after.
3. Pointcut: A predicate that matches join points (specific method executions where advice should run).
4. Join Point: A point during execution, such as a method call, where an advice can be applied.
5. Weaving: Linking aspects with the main application logic at runtime or compile-time.

Example: Logging Using AOP

1. Add Dependency (if not already added)

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

2. Create an Aspect

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class LoggingAspect {

    @Before("execution(* com.example.demo.service.*.*(..))")
    public void logBeforeMethodExecution() {
        System.out.println("Logging before method execution...");
    }
}
```

3. Explanation of the Code

- @Aspect: Marks the class as an aspect.
- @Before: This advice runs before methods are executed.
- execution(* com.example.demo.service.*.*(..)): Defines a pointcut to match all methods in the service package.

4. Service Class (Business Logic)

```
package com.example.demo.service;

import org.springframework.stereotype.Service;

@Service
public class MyService {
    public void performTask() {
        System.out.println("Performing task...");
    }
}
```

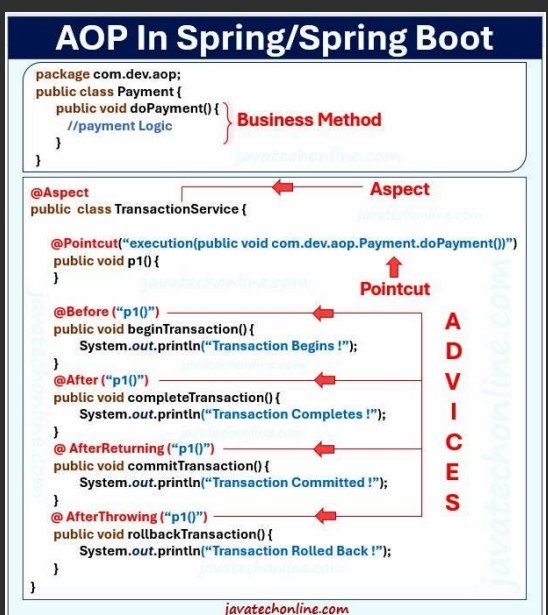
5. Run the Application

When you call performTask(), the output will be:

```
Logging before method execution...
Performing task...
```

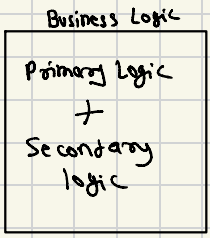
Benefits of AOP:

- Centralized handling of cross-cutting concerns.
- Cleaner and more maintainable code.
- Dynamic addition of behavior without modifying existing logic.



* AOP is methodology of programming to solve the problem. Lec2
* AOP is build on top of OOPs style programming.

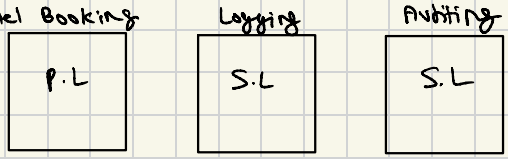
* Store data into DB → primary logic (mandatory)
* print logs → secondary logic (optional logic)



learn Design principle to segregate P.L and S.L.
OR we can give S.L. to AOP to handle it.

* Logging → keep track of the code flow [log4j, SLF4J]
* Auditing → keep track of activity flow
  - eg. Hotel Booking → save, fetch, delete

* OOPS
  - clumsy code due to primary logic and secondary logic.
  - No reusability
  - kills the readability.
  - No clean code.
  - Debugging will be complex.
  - Not recommended for large scale application.



* The framework that are based on AOP principles r/as AOP enabled framework. eg:- Spring AOP
* AOP principles / terminologies are:-
  1) Aspect
  2) Advice
  3) Joinpoint
  4) pointcut
  5) Target

* Ascept : It is a class that represent secondary logic.
  ↳ What you want apply.
* Advice : It is the action taken by Ascept.
  ↳ How you want to apply.
    ↳ Types
      ↳ Before Advice
      ↳ After Advice
      ↳ Around Advice
      ↳ Throws Advice
* Joinpoints : place in target class
  ↳ where we can apply
* point cut : collection of points where the aspect are advised
  ↳ where we have applied / want to apply

1. **Aspect**: It is a class that represents secondary logic or cross-cutting concerns. It defines *what* you want to apply, such as logging, security, or transactions.
2. **Advice**: It is the action taken by the Aspect. It specifies *how* you want to apply the aspect.
   - **Types of Advice**:
     - **Before Advice**: Executes before the joinpoint.
     - **After Advice**: Executes after the joinpoint.
     - **Around Advice**: Surrounds the joinpoint and can control whether the joinpoint is executed.
     - **Throws Advice**: Executes when a method throws an exception.
3. **Joinpoint**: A point in the target class where the aspect can be applied, such as method execution or object creation.
4. **Pointcut**: A collection of joinpoints where aspects are applied or intended to be applied. It defines *where* the advice should be executed.

```
com.example.projectname
├── config              # Configuration classes
│   └── WebConfig.java   # Web-related configuration (CORS, interceptors, etc.)
│   └── AppConfig.java   # Application-specific configurations
├── controller          # REST API controllers (HTTP layer)
│   └── UserController.java
├── dto                 # Data Transfer Objects for API communication
│   └── UserDTO.java
├── service             # Business logic layer
│   ├── UserService.java # Interface for user service
│   ├── impl            # Implementation of services
│   │   └── UserServiceImpl.java
├── dao                 # Data Access Objects for database interaction
│   └── UserDAO.java
├── entity              # JPA Entities representing database tables
│   └── User.java
├── exception           # Custom exceptions and handlers
│   ├── UserNotFoundException.java
│   └── GlobalExceptionHandler.java
├── util                # Utility classes for validations, conversions, etc.
│   ├── ValidationUtil.java
│   └── UserMapper.java
├── security            # Security-related configurations and classes
│   ├── SecurityConfig.java
│   ├── JwtUtil.java
│   └── AuthEntryPoint.java
├── repository          # Repository interfaces (if different from DAO)
│   └── UserRepository.java
├── model               # Additional data models (optional, for non-entity objects)
│   └── LoginRequest.java
├── bootstrap           # Data initialization scripts or classes
│   └── DataLoader.java
├── tests               # Test-related files
│   ├── unit            # Unit tests
│   │   └── UserServiceTest.java
│   ├── integration     # Integration tests
│   │   └── UserControllerIT.java
├── application.properties # Configuration properties file
```

1. DTO (Data Transfer Object):

- **Purpose**: Used to transfer data between layers (e.g., Controller and Service layers).
- **Key Benefit**: Keeps the data structure clean and reduces unnecessary exposure of the domain model.

2. DAO (Data Access Object):

- **Purpose**: Encapsulates database interaction logic (e.g., CRUD operations).
- **Key Benefit**: Abstracts persistence logic from the rest of the application.

1.Entity (Database Model)

```
package com.example.employeemanagement.entity;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;

@Entity
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String department;
    private Double salary;

    // Getters and Setters
}
```

2. DTO (Data Transfer Object)

```
package com.example.employeemanagement.dto;

public class EmployeeDTO {
    private String name;
    private String department;
    private Double salary;

    // Constructor, Getters, and Setters
    public EmployeeDTO(String name, String department, Double salary) {
        this.name = name;
        this.department = department;
        this.salary = salary;
    }
}
```

3. DAO (Database Access Layer)

```
package com.example.employeemanagement.dao;

import com.example.employeemanagement.entity.Employee;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface EmployeeDAO extends JpaRepository<Employee, Long> {
}
```

4. Controller Layer (API Endpoints)

```
package com.example.employeemanagement.controller;

import com.example.employeemanagement.dto.EmployeeDTO;
import com.example.employeemanagement.service.EmployeeService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/employees")
public class EmployeeController {

    @Autowired
    private EmployeeService employeeService;

    @PostMapping
    public String addEmployee(@RequestBody EmployeeDTO employeeDTO) {
        employeeService.addEmployee(employeeDTO);
        return "Employee added successfully!";
    }

    @GetMapping
    public List<EmployeeDTO> getAllEmployees() {
        return employeeService.getAllEmployees();
    }

    @PutMapping("/{id}")
    public String updateEmployee(@PathVariable Long id, @RequestBody EmployeeDTO employeeDTO) {
        employeeService.updateEmployee(id, employeeDTO);
        return "Employee updated successfully!";
    }

    @DeleteMapping("/{id}")
    public String deleteEmployee(@PathVariable Long id) {
        employeeService.deleteEmployee(id);
        return "Employee deleted successfully!";
    }
}
```

5. Service Layer (Business Logic)

```
package com.example.employeemanagement.service;

import com.example.employeemanagement.dao.EmployeeDAO;
import com.example.employeemanagement.dto.EmployeeDTO;
import com.example.employeemanagement.entity.Employee;
import com.example.employeemanagement.exception.EmployeeNotFoundException;
import com.example.employeemanagement.util.EmployeeMapper;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.stream.Collectors;

@Service
public class EmployeeService {
    @Autowired
    private EmployeeDAO employeeDAO;

    @Autowired
    private EmployeeMapper employeeMapper;

    // Create a new employee
    public void addEmployee(EmployeeDTO employeeDTO) {
        Employee employee = employeeMapper.toEntity(employeeDTO);
        employeeDAO.save(employee);
    }

    // Get all employees
    public List<EmployeeDTO> getAllEmployees() {
        return employeeDAO.findAll()
                .stream()
                .map(employeeMapper::toDTO)
                .collect(Collectors.toList());
    }

    // Update an employee
    public void updateEmployee(Long id, EmployeeDTO employeeDTO) {
        Employee existingEmployee = employeeDAO.findById(id)
                .orElseThrow(() -> new EmployeeNotFoundException("Employee not found"));
        existingEmployee.setName(employeeDTO.getName());
        existingEmployee.setDepartment(employeeDTO.getDepartment());
        existingEmployee.setSalary(employeeDTO.getSalary());
        employeeDAO.save(existingEmployee);
    }

    // Delete an employee
    public void deleteEmployee(Long id) {
        if (!employeeDAO.existsById(id)) {
            throw new EmployeeNotFoundException("Employee not found");
        }
        employeeDAO.deleteById(id);
    }
}
```

Continue

6. Utilities (Reusable Classes)

EmployeeMapper: For Entity-DTO Conversion

```
package com.example.employeemanagement.util;

import com.example.employeemanagement.dto.EmployeeDTO;
import com.example.employeemanagement.entity.Employee;
import org.springframework.stereotype.Component;

@Component
public class EmployeeMapper {
    public Employee toEntity(EmployeeDTO employeeDTO) {
        Employee employee = new Employee();
        employee.setName(employeeDTO.getName());
        employee.setDepartment(employeeDTO.getDepartment());
        employee.setSalary(employeeDTO.getSalary());
        return employee;
    }

    public EmployeeDTO toDTO(Employee employee) {
        return new EmployeeDTO(employee.getName(), employee.getDepartment(), employee.getSalary());
    }
}
```

ValidationUtil: For Input Validations

```
package com.example.employeemanagement.util;

public class ValidationUtil {

    public static boolean isValidSalary(Double salary) {
        return salary != null && salary > 0;
    }
}
```

7. Custom Exceptions

```
package com.example.employeemanagement.exception;

public class EmployeeNotFoundException extends RuntimeException {
    public EmployeeNotFoundException(String message) {
        super(message);
    }
}
```

GlobalExceptionHandler

```
package com.example.employeemanagement.exception;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;

@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(EmployeeNotFoundException.class)
    public ResponseEntity<String> handleEmployeeNotFoundException(EmployeeNotFoundException ex) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(ex.getMessage());
    }
}
```

8. application.properties

```
spring.datasource.url=jdbc:mysql://localhost:3306/employeedb
spring.datasource.username=root
spring.datasource.password=root
spring.jpa.hibernate.ddl-auto=update
server.port=8080
```

Testing the Application

1. **Add Employee**:
`POST /employees`

```
json

{
    "name": "John Doe",
    "department": "IT",
    "salary": 5000
}
```

2. **Get All Employees**:
`GET /employees`
Response:

```
json

[
    {
        "name": "John Doe",
        "department": "IT",
        "salary": 5000
    }
]
```

3. **Update Employee**:
`PUT /employees/1`

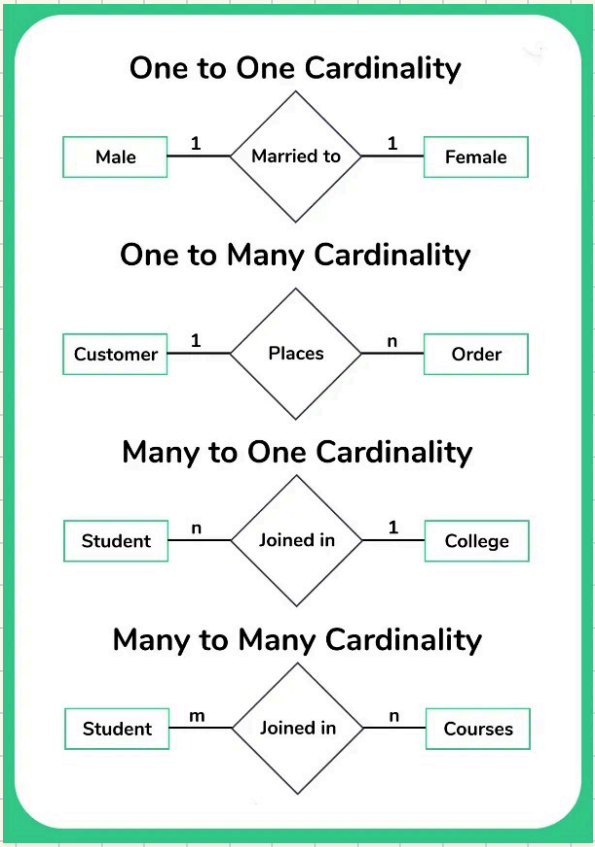
```
json

{
    "name": "Jane Doe",
    "department": "HR",
    "salary": 6000
}
```

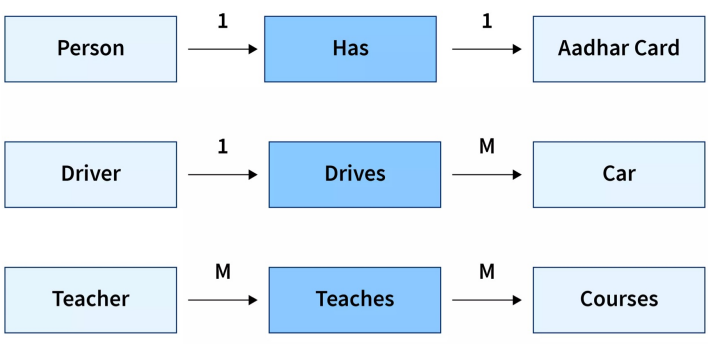
4. **Delete Employee**:
`DELETE /employees/1`

```
com.example.employeemanagement
├── config              # Configuration classes (if needed)
│   └── AppConfig.java
├── controller          # REST API controllers (HTTP endpoints)
│   └── EmployeeController.java
├── dto                 # Data Transfer Objects for API communication
│   └── EmployeeDTO.java
├── dao                 # Data Access Objects for database interaction
│   └── EmployeeDAO.java
├── entity              # JPA Entities representing database tables
│   └── Employee.java
├── service             # Business logic layer
│   └── EmployeeService.java
├── util                # Utility classes for validations, conversions, etc.
│   ├── EmployeeMapper.java
│   └── ValidationUtil.java
├── exception           # Custom exceptions and handlers
│   ├── EmployeeNotFoundException.java
│   └── GlobalExceptionHandler.java
└── application.properties # Configuration file
```

Relationships in DBMS

Lec 3





SCALER Topics

1NF (First Normal Form)

Rule: Eliminate repeating groups; each column should contain atomic values (no multiple values in one cell).

Example Table (Non-1NF):

StudentID	Name	Subjects
1	Alice	Math, Science
2	Bob	English, History

Converted to 1NF:

StudentID	Name	Subject
1	Alice	Math
1	Alice	Science
2	Bob	English
2	Bob	History

2NF (Second Normal Form)

Rule: Must be in 1NF, and all non-prime attributes (non-key columns) must depend on the entire primary key (no partial dependency).

Example Table (Non-2NF):

StudentID	CourseID	CourseName	Instructor
1	101	Math	Dr. Smith
2	102	Science	Dr. Johnson

Here, `CourseName` depends only on `CourseID`, not on the full primary key (`StudentID` + `CourseID`).

Converted to 2NF:

- Student-Course Table:

StudentID	CourseID
1	101
2	102

- Course Table:

CourseID	CourseName	Instructor
101	Math	Dr. Smith
102	Science	Dr. Johnson

3NF (Third Normal Form)

Rule: Must be in 2NF, and no transitive dependency (non-prime attributes should depend only on the primary key, not on other non-prime attributes).

Example Table (Non-3NF):

CourseID	CourseName	Instructor	InstructorDept
101	Math	Dr. Smith	Mathematics
102	Science	Dr. Johnson	Physics

Here, `InstructorDept` depends on `Instructor`, not directly on `CourseID`.

Converted to 3NF:

- Course Table:

CourseID	CourseName	Instructor
101	Math	Dr. Smith
102	Science	Dr. Johnson

- Instructor Table:

Instructor	InstructorDept
Dr. Smith	Mathematics
Dr. Johnson	Physics

BCNF (Boyce-Codd Normal Form)

Rule: Must be in 3NF, and every determinant must be a candidate key.

Example Table (Non-BCNF):

CourseID	Instructor	Room
101	Dr. Smith	Room A
101	Dr. Brown	Room B

Here, `Room` depends on both `CourseID` and `Instructor`, but `Instructor` itself can determine `Room`.

Converted to BCNF:

- Course-Instructor Table:

CourseID	Instructor
101	Dr. Smith
101	Dr. Brown

- Instructor-Room Table:

Instructor	Room
Dr. Smith	Room A
Dr. Brown	Room B

4NF (Fourth Normal Form)

Rule: Must be in BCNF, and no multivalued dependencies.

Example Table (Non-4NF):

StudentID	CourseID	Hobby
1	101	Swimming
1	101	Painting
1	102	Swimming
1	102	Painting

Here, `CourseID` and `Hobby` are independent of each other for a given `StudentID`.

Converted to 4NF:

- Student-Course Table:

StudentID	CourseID
1	101
1	102

- Student-Hobby Table:

StudentID	Hobby
1	Swimming
1	Painting

5NF (Fifth Normal Form)

Rule: Must be in 4NF, and every join dependency should be implied by candidate keys.

Example Table (Non-5NF):

StudentID	CourseID	Instructor
1	101	Dr. Smith
1	101	Dr. Brown
2	102	Dr. Johnson

Here, splitting the table into smaller ones might lead to loss of information during joins.

Converted to 5NF:

- Student-Course Table:

StudentID	CourseID
1	101
2	102

- Course-Instructor Table:

CourseID	Instructor
101	Dr. Smith
101	Dr. Brown
102	Dr. Johnson

- Student-Instructor Table:

StudentID	Instructor
1	Dr. Smith
1	Dr. Brown
2	Dr. Johnson

Summary:

- **1NF**: Eliminate repeating groups.
- **2NF**: Eliminate partial dependency.
- **3NF**: Eliminate transitive dependency.
- **BCNF**: Every determinant is a candidate key.
- **4NF**: Eliminate multivalued dependencies.
- **5NF**: Eliminate redundancy caused by join dependencies.

Normalization in DBMS

1. Definition:

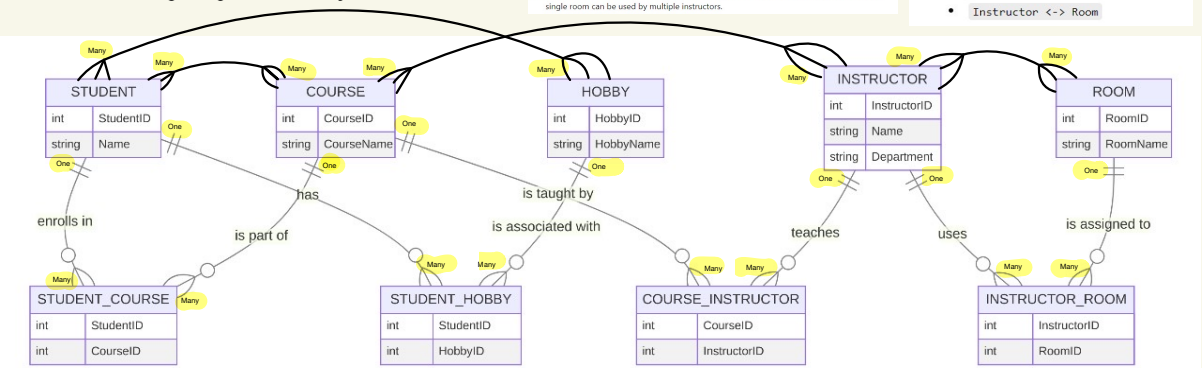
A process of organizing database structure to reduce redundancy and improve data integrity.

2. Goals:
   - Minimize data redundancy.
   - Ensure data consistency.
   - Simplify database maintenance.
3. Core Concepts:
   - Functional Dependency: One attribute uniquely determines another.
   - Primary Key: A unique identifier for table rows.
4. Normal Forms (NFs):
   - 1NF: Atomic values; no repeating groups.
   - 2NF: Achieves 1NF; eliminates partial dependencies.
   - 3NF: Achieves 2NF; removes transitive dependencies.
   - BCNF: Stricter 3NF; every determinant is a candidate key.
   - 4NF: Removes multi-valued dependencies.
   - 5NF: Resolves join dependencies.
5. Advantages:
   - Reduces data redundancy.
   - Improves data integrity.
   - Enhances database performance for updates.
6. Disadvantages:
   - Increased number of tables.
   - Complex queries due to joins.

1. **Student to Student-Course**: A single student can enroll in multiple courses.
2. **Course to Student-Course**: A single course can have multiple students enrolled.
3. **Course to Course-Instructor**: A single course can be taught by multiple instructors.
4. **Instructor to Course-Instructor**: A single instructor can teach multiple courses.
5. **Student to Student-Hobby**: A single student can have multiple hobbies.
6. **Hobby to Student-Hobby**: A single hobby can be associated with multiple students.
7. **Instructor to Instructor-Room**: A single instructor can use multiple rooms.
8. **Room to Instructor-Room**: A single room can be used by multiple instructors.
9. **Student to Course (via Student-Course)**: A single student can enroll in multiple courses, and a single course can have multiple students.
10. **Course to Instructor (via Course-Instructor)**: A single course can be taught by multiple instructors, and a single instructor can teach multiple courses.
11. **Student to Hobby (via Student-Hobby)**: A single student can have multiple hobbies, and a single hobby can be shared by multiple students.
12. **Instructor to Room (via Instructor-Room)**: A single instructor can use multiple rooms, and a single room can be used by multiple instructors.

Summary of Relationships:

- **One-to-Many**:
  - `Student -> Student-Course`
  - `Course -> Student-Course`
  - `Course -> Course-Instructor`
  - `Instructor -> Course-Instructor`
  - `Student -> Student-Hobby`
  - `Hobby -> Student-Hobby`
  - `Instructor -> Instructor-Room`
  - `Room -> Instructor-Room`
- **Many-to-Many**:
  - `Student <-> Course`
  - `Course <-> Instructor`
  - `Student <-> Hobby`
  - `Instructor <-> Room`



ER Diagram

Spring boot with JPA annotations

Entity Classes & Mappings

Student

```java
@Entity
public class Student {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long studentId;

    private String name;

    @OneToMany(mappedBy = "student")
    private List<StudentCourse> studentCourses;

    @OneToMany(mappedBy = "student")
    private List<StudentHobby> studentHobbies;

    // Getters and Setters
}
```

Instructor

```java
@Entity
public class Instructor {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long instructorId;

    private String name;

    private String department;

    @OneToMany(mappedBy = "instructor")
    private List<CourseInstructor> courseInstructors;

    @OneToMany(mappedBy = "instructor")
    private List<InstructorRoom> instructorRooms;

    // Getters and Setters
}
```

Course

```java
@Entity
public class Course {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long courseId;

    private String courseName;

    @OneToMany(mappedBy = "course")
    private List<StudentCourse> studentCourses;

    @OneToMany(mappedBy = "course")
    private List<CourseInstructor> courseInstructors;

    // Getters and Setters
}
```

Hobby

```java
@Entity
public class Hobby {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long hobbyId;

    private String hobbyName;

    @OneToMany(mappedBy = "hobby")
    private List<StudentHobby> studentHobbies;

    // Getters and Setters
}
```

Room

```java
@Entity
public class Room {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long roomId;

    private String roomName;

    @OneToMany(mappedBy = "room")
    private List<InstructorRoom> instructorRooms;

    // Getters and Setters
}
```

CourseInstructor

```java
@Entity
public class CourseInstructor {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne
    @JoinColumn(name = "course_id")
    private Course course;

    @ManyToOne
    @JoinColumn(name = "instructor_id")
    private Instructor instructor;

    // Getters and Setters
}
```

InstructorRoom

```java
@Entity
public class InstructorRoom {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne
    @JoinColumn(name = "instructor_id")
    private Instructor instructor;

    @ManyToOne
    @JoinColumn(name = "room_id")
    private Room room;

    // Getters and Setters
}
```

Relationship Tables (Many-to-Many)

StudentCourse

```java
@Entity
public class StudentCourse {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne
    @JoinColumn(name = "student_id")
    private Student student;

    @ManyToOne
    @JoinColumn(name = "course_id")
    private Course course;

    // Getters and Setters
}
```

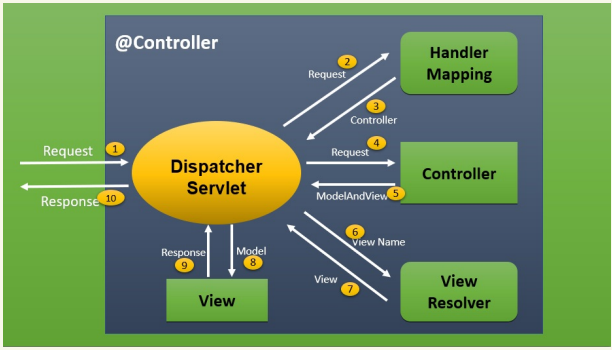
StudentHobby

```java
@Entity
public class StudentHobby {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne
    @JoinColumn(name = "student_id")
    private Student student;

    @ManyToOne
    @JoinColumn(name = "hobby_id")
    private Hobby hobby;

    // Getters and Setters
}
```

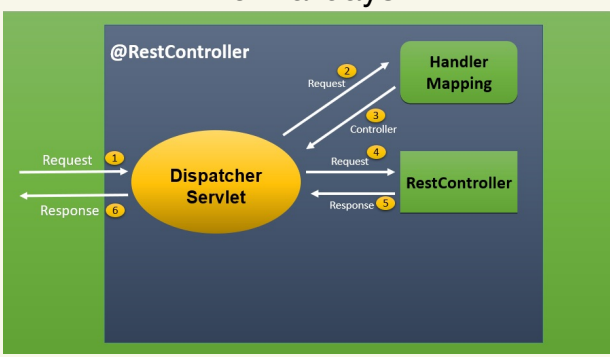
1. Overview of Spring MVC

- Model-View-Controller (MVC): A design pattern to separate application logic (Model), user interface (View), and control flow (Controller).
- Spring MVC is part of the Spring Framework, enabling the development of flexible and loosely coupled web applications.

Old



Now-a-days



2. Key Components of Spring MVC

1. Model
   - Encapsulates the application data and business logic.
   - Represents the state and is typically implemented using POJOs (Plain Old Java Objects).
   - Works with persistence frameworks like JPA or Hibernate to interact with databases.
2. View
   - Represents the user interface (UI) layer.
   - Displays data to the user and accepts inputs.
   - Supports various rendering technologies like Thymeleaf, JSP, FreeMarker, and Mustache.
3. Controller
   - Handles incoming HTTP requests, processes user input, and delegates tasks to the model.
   - Acts as the intermediary between the Model and View.

3. Request Flow in Spring MVC

1. User Request: A user sends an HTTP request (e.g., GET or POST).
2. DispatcherServlet: Central servlet in Spring MVC that intercepts all requests.
3. Handler Mapping: Maps the request to the appropriate controller based on the URL.
4. Controller: Processes the request, interacts with the model, and prepares data for the view.
5. Model: Business logic is applied, and data is fetched/processed.
6. ViewResolver: Determines which view (HTML, JSON, etc.) to render based on the controller's response.
7. Response: The rendered view or output is sent back as the HTTP response.

4. Key Annotations in Spring MVC

- @Controller: Defines a class as a controller.
- @RestController: Combines @Controller and @ResponseBody for RESTful APIs.
- @RequestMapping: Maps HTTP requests to controller methods (can be replaced with @GetMapping, @PostMapping, etc.).
- @ModelAttribute: Binds form data or request parameters to an object.
- @ResponseBody: Returns data (e.g., JSON or XML) directly as the HTTP response.
- @PathVariable: Captures dynamic values from the URL.
- @RequestParam: Binds query parameters to method arguments.

5. Spring MVC Architecture

- Built around DispatcherServlet, which acts as the front controller.
- Works with components like:
  - HandlerMapping: Matches requests to controllers.
  - ViewResolver: Resolves view templates (e.g., .html, .jsp).
  - ModelAndView: Combines model data and view information.

6. Advantages of Spring MVC

- Separation of concerns: Clear division of responsibilities.
- Flexible view technologies: Supports various front-end tools.
- Easy integration: Works seamlessly with other Spring projects (e.g., Spring Boot, Spring Security).
- Testability: Loose coupling improves unit testing.

7. DispatcherServlet Lifecycle

1. Initialization: Loads configuration (e.g., web.xml or Spring Boot's application.properties).
2. Request Handling: Delegates tasks to controllers and returns a response.
3. Cleanup: Releases resources after processing requests.

This is a high-level overview of Spring MVC. Let me know if you'd like details on specific components or examples!

Spring Security

lec 1

1. Overview

- Spring Security is a framework for authentication, authorization, and protection against security threats.
- Easily integrates with Spring Boot.

2. Core Concepts

- Authentication: Verifies the identity of a user (e.g., login).
- Authorization: Controls access to resources based on roles/permissions.
- Security Context: Stores authentication details (via SecurityContextHolder).

3. Key Features

1. Authentication Mechanisms:
   - Username/password, JWT, OAuth2, LDAP, SAML, etc.
2. Authorization:
   - Role-based or method-level using annotations.
3. Session Management:
   - Protects against session fixation and manages concurrent sessions.
4. Built-in Filters:
   - Preconfigured filter chain (e.g., CsrfFilter, AuthenticationFilter).
5. Password Encoding:
   - Secure password storage (e.g., BCrypt).

4. Architecture

- Security Filters: Chain of filters processes requests.
- AuthenticationManager: Handles user authentication.
- AccessDecisionManager: Enforces access rules.

5. Annotations

- @EnableWebSecurity: Enables Spring Security.
- @PreAuthorize: Secures methods before execution.
- @Secured or @RolesAllowed: Role-based access control.
- @PostAuthorize: Secures methods after execution.

6. Default Security (Spring Boot)

- Auto-configured login form and default user/password.
- Can be customized via SecurityFilterChain.

7. Configuration Example

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http.csrf().disable()
        .authorizeRequests()
        .antMatchers("/admin/**").hasRole("ADMIN")
        .anyRequest().authenticated()
        .and().formLogin();
    return http.build();
}
```

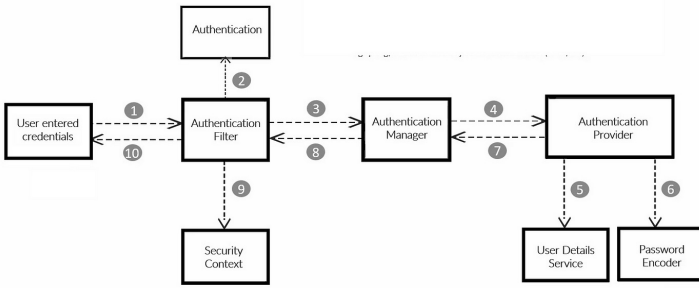
8. Common Threats Handled

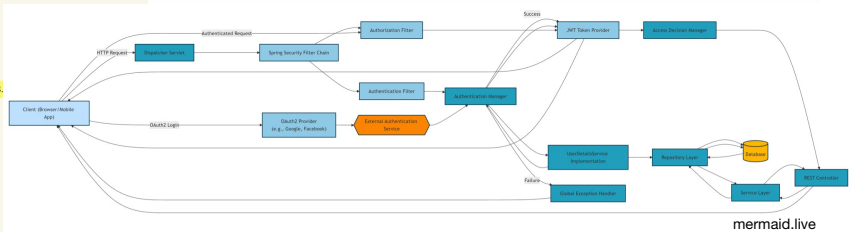
- CSRF: Prevents unauthorized actions by trusted users.
- XSS: Escapes output to prevent script injection.
- Session Fixation: Protects against hijacked sessions.

9. Advantages

- Comprehensive and customizable.
- Supports modern authentication (e.g., OAuth2, JWT).
- Works seamlessly with Spring Boot and Spring projects.

Spring Security Flow





CORS (Cross-Origin Resource Sharing)

CORS is about allowing or restricting access to resources from a different domain.

CORS ka use tab hota hai jab ek website ke resources ko kisi doosri domain wali website se access karne ki permission deni ya rokni ho.

Real-Life Example:

1. Scenario:
   - A front-end application hosted at https://frontend.com tries to fetch data from an API at https://api.server.com.
2. CORS Policy:
   - If the API server does not allow requests from https://frontend.com, the browser blocks the request.
3. Use Case:
   - You enable CORS in the API server to allow specific domains to access its resources safely (e.g., only allow trusted front-end applications).

localhost:8080/home
localhost:8080/getBalance

CSRF (Cross-Site Request Forgery)

CSRF is about protecting a server from unauthorized actions made by a malicious third-party site on behalf of an authenticated user.

CSRF ka use tab hota hai jab ek malicious website kisi asli website par logged-in user ke behalf par bina uske consent ke koi unauthorized action perform karne ki koshish kare. Isse bachne ke liye CSRF tokens ka istemal hota hai.

For example:
Agar aap bank.com par logged in hain aur ek malicious website aapke account se fund transfer karne ki koshish kare, toh CSRF token verify karke bank aise requests ko reject kar deta hai.

Real-Life Example:

1. Scenario:
   - A user is logged into https://bank.com.
   - A malicious site tricks the user into clicking a link that makes a POST request to https://bank.com/transfer?amount=1000&to=attacker.
2. CSRF Protection:
   - The bank's server uses CSRF tokens to ensure requests come from its own site, not an external malicious one.
   - If the request lacks a valid CSRF token, it gets rejected.

Key Difference:

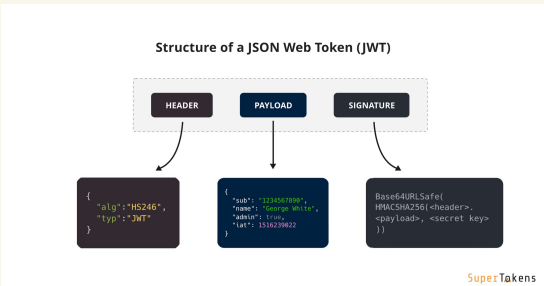
- CORS: Protects cross-domain resource sharing.
- CSRF: Prevents unauthorized actions on a user's behalf.

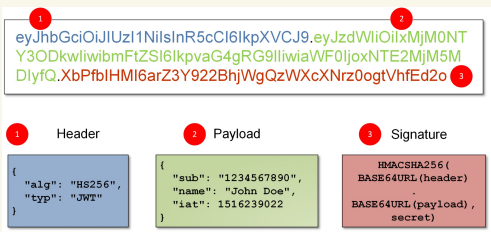
Spring Security with JWT

Lec 1

Spring Security with JWT – Key Points

1. **What is JWT?**
   - JSON Web Token is a compact, self-contained token for authentication.
   - Contains user details, roles, and expiry info.
2. **Why JWT?**
   - Stateless authentication (no session storage needed).
   - Ideal for REST APIs and scalable systems.
3. **JWT Structure**:
   - `Header.Payload.Signature`
   - Example: `eyJhbGciOiJIUzI1Ni...abc123`
4. **Workflow**:
   - **Login**: User sends credentials.
   - **Token Creation**: Server generates and returns JWT on successful authentication.
   - **Token Usage**: Client sends JWT in the `Authorization` header (`Bearer <JWT>`).
   - **Validation**: Server validates the token for each request.
5. **Advantages**:
   - No need to maintain session state.
   - Reduces database calls.
   - Works across distributed systems (e.g., microservices).
6. **Implementation Steps**:
   - Add dependencies (`spring-security`, `jjwt`).
   - Create a **JWT utility** for token generation and validation.
   - Add a **filter** to validate tokens for incoming requests.
   - Configure **Spring Security** to integrate JWT.





This approach ensures secure, stateless authentication for modern applications.

pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.11.5</version>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <version>0.11.5</version>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId>
    <version>0.11.5</version>
</dependency>
```

Create JWT Utility Class

```
import io.jsonwebtoken.*;
import org.springframework.stereotype.Component;

import java.util.Date;

@Component
public class JwtUtil {

    private final String SECRET_KEY = "my_secret_key";

    // Generate JWT
    public String generateToken(String username) {
        return Jwts.builder()
                .setSubject(username) // Add username
                .setIssuedAt(new Date()) // Add issue time
                .setExpiration(new Date(System.currentTimeMillis() + 1000 * 60 * 60)) // 1-hour expiry
                .signWith(SignatureAlgorithm.HS256, SECRET_KEY) // Sign with secret key
                .compact();
    }

    // Validate JWT
    public boolean validateToken(String token, String username) {
        String extractedUsername = extractUsername(token);
        return extractedUsername.equals(username) && !isTokenExpired(token);
    }

    // Extract username from JWT
    public String extractUsername(String token) {
        return Jwts.parser().setSigningKey(SECRET_KEY).parseClaimsJws(token).getBody().getSubject();
    }

    // Check token expiration
    private boolean isTokenExpired(String token) {
        Date expiration = Jwts.parser().setSigningKey(SECRET_KEY).parseClaimsJws(token).getBody().getExpiration();
        return expiration.before(new Date());
    }
}
```

Create REST Controller

```
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/api")
public class AuthController {

    private final JwtUtil jwtUtil;

    public AuthController(JwtUtil jwtUtil) {
        this.jwtUtil = jwtUtil;
    }

    // Login endpoint
    @PostMapping("/login")
    public String login(@RequestParam String username, @RequestParam String password) {
        // Hardcoded credentials (replace with DB check in real apps)
        if ("user".equals(username) && "password".equals(password)) {
            // Generate and return token
            return jwtUtil.generateToken(username);
        }
        throw new RuntimeException("Invalid username or password");
    }
}
```

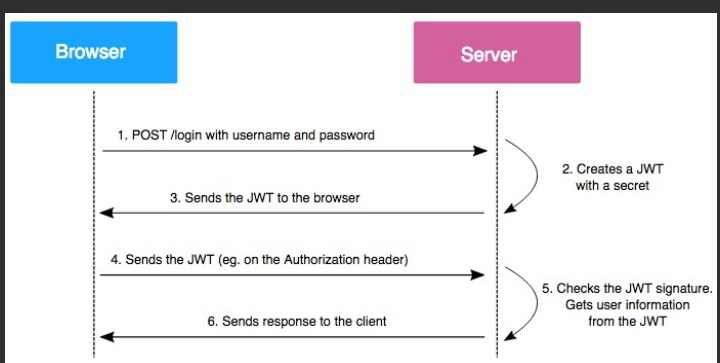
Secure API Endpoints

```
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;

import javax.servlet.FilterChain;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@Component
public class JwtFilter extends OncePerRequestFilter {

    private final JwtUtil jwtUtil;

    public JwtFilter(JwtUtil jwtUtil) {
        this.jwtUtil = jwtUtil;
    }

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain chain)
            throws java.io.IOException, javax.servlet.ServletException {

        String authHeader = request.getHeader("Authorization");
        if (authHeader != null && authHeader.startsWith("Bearer ")) {
            String token = authHeader.substring(7); // Extract token
            String username = jwtUtil.extractUsername(token);

            if (username != null && jwtUtil.validateToken(token, username)) {
                SecurityContextHolder.getContext().setAuthentication(
                        new UsernamePasswordAuthenticationToken(username, null, null)
                );
            }
        }
        chain.doFilter(request, response);
    }
}
```

Configure Spring Security

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.web.SecurityFilterChain;
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

@Configuration
public class SecurityConfig {

    private final JwtFilter jwtFilter;

    public SecurityConfig(JwtFilter jwtFilter) {
        this.jwtFilter = jwtFilter;
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http.csrf().disable()
            .authorizeRequests()
            .antMatchers("/api/login").permitAll() // Allow login endpoint
            .anyRequest().authenticated() // Secure other endpoints
            .and()
            .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS); // Stateless sessions

        http.addFilterBefore(jwtFilter, UsernamePasswordAuthenticationFilter.class); // Add JWT filter
        return http.build();
    }
}
```

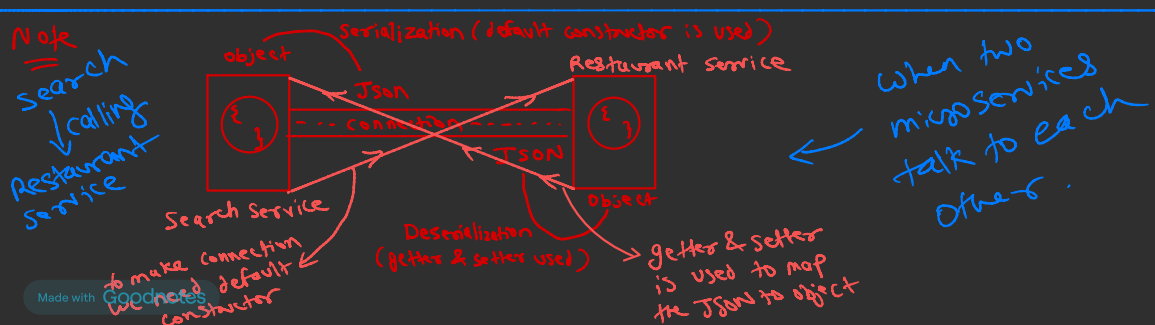
Test the Application



1. **Login**:
   - Make a POST request to `/api/login` with `username=user` and `password=password`.
   - Receive a JWT token.
2. **Access Secured Endpoint**:
   - Include the token in the `Authorization` header:

Authorization: Bearer <your-token>

Access other endpoints securely.



Lec 1

Docker

1. **What is Docker?**
   - Docker is a containerization platform that packages applications and their dependencies into lightweight, portable containers.
   - Containers ensure consistent execution across different environments (development, testing, production).

2. **Why Use Docker with Spring Boot?**
   - **Simplified Deployment**: Run Spring Boot apps consistently across machines.
   - **Portability**: Works the same on local systems, CI/CD pipelines, and cloud platforms.
   - **Resource Efficiency**: Containers use fewer resources than virtual machines.
   - **Microservices-Friendly**: Each Spring Boot service can run in its own container.

3. **Key Concepts in Docker**
   - **Image**: A lightweight, standalone, and executable package containing everything to run the app (Spring Boot JAR + OS dependencies).
   - **Container**: A running instance of an image.
   - **Dockerfile**: A script to define how a Docker image is built.
   - **Docker Hub**: A registry to store and share Docker images.

4. **Steps to Dockerize a Spring Boot Application**
   1. **Create a Spring Boot JAR**:
      - Use `mvn package` or `gradle build` to generate the JAR file.
   2. **Write a Dockerfile**:
      - Define the base image, add the JAR file, and specify how to run the app.
   3. **Build Docker Image**:
      - Use `docker build` command to create the image.
   4. **Run Docker Container**:
      - Use `docker run` to start a container from the image.

5. **Sample Dockerfile for Spring Boot**

dockerfile — Copy

```
# Use a lightweight base image
FROM openjdk:17-jdk-slim

# Set working directory inside the container
WORKDIR /app

# Copy the Spring Boot JAR file into the container
COPY target/my-spring-boot-app.jar app.jar

# Expose the application port
EXPOSE 8080

# Command to run the application
ENTRYPOINT ["java", "-jar", "app.jar"]
```

6. **Important Docker Commands**
   - `docker build -t spring-boot-app .` – Build an image from the Dockerfile.
   - `docker run -p 8080:8080 spring-boot-app` – Run the container and map port 8080.
   - `docker ps` – List running containers.
   - `docker stop <container_id>` – Stop a container.

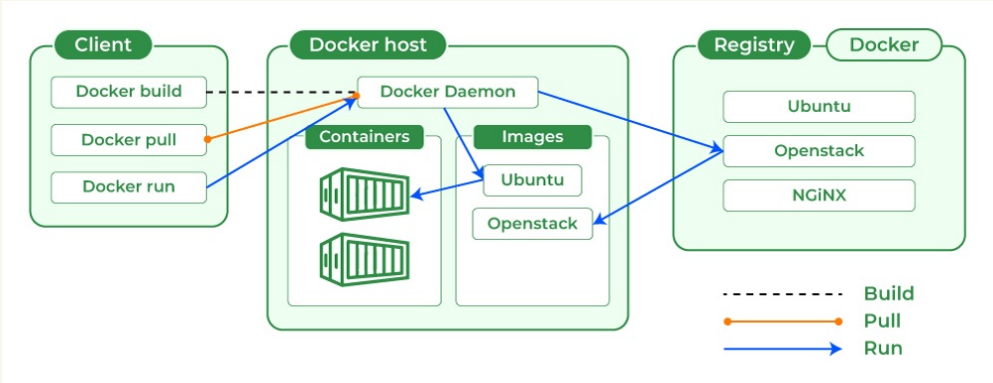
7. **Advantages of Docker in Spring Boot**
   - **Environment Isolation**: Avoid dependency conflicts.
   - **Ease of Scaling**: Easily scale Spring Boot microservices with multiple containers.
   - **CI/CD Integration**: Simplifies automated builds, tests, and deployments.

8. **Deploying Spring Boot with Docker Compose**
   - Use `docker-compose.yml` to define and manage multiple services (e.g., app + database).
   - Example:

yaml — Copy

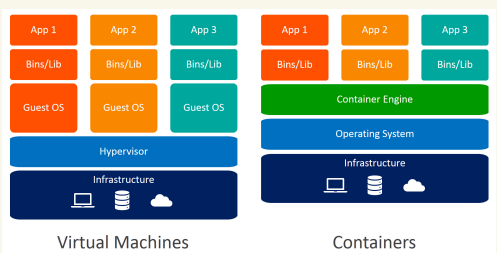
```
version: "3.8"
services:
  app:
    build: .
    ports:
      - "8080:8080"
  db:
    image: postgres
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
```

Architecture of Docker



Explanation of Docker Workflow (Based on the Diagram):

1. Client:
   - The user interacts with Docker using commands like docker build, docker pull, and docker run.
2. Docker Host:
   - Docker Daemon:
     - A background service that handles the building, pulling, and running of Docker images and containers.
   - Images:
     - Pre-built application templates like Ubuntu, OpenStack, or Nginx.
     - Images are the foundation to create containers.
   - Containers:
     - Running instances of images. Each container is isolated and runs independently.
3. Registry (Docker Hub):
   - A centralized repository that stores Docker images.
   - Examples of images include Ubuntu, OpenStack, or custom-built images.
4. Workflow:
   - Build:
     - The client uses docker build to create a custom image. The Docker Daemon builds the image locally.
   - Pull:
     - The client uses docker pull to download an image from the Docker registry.
   - Run:
     - The client uses docker run to create and run a container from an image.
5. Data Flow:
   - Images are pulled from the registry to the Docker host.
   - Containers are created from these images on the Docker host.
   - Containers execute the application or service defined in the image.



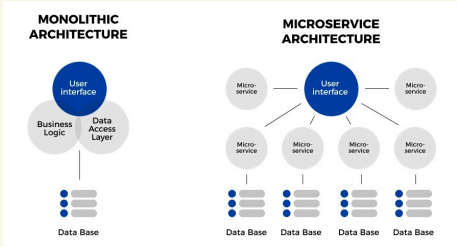
Servers managed by AWS
Containers managed by Kubernetes

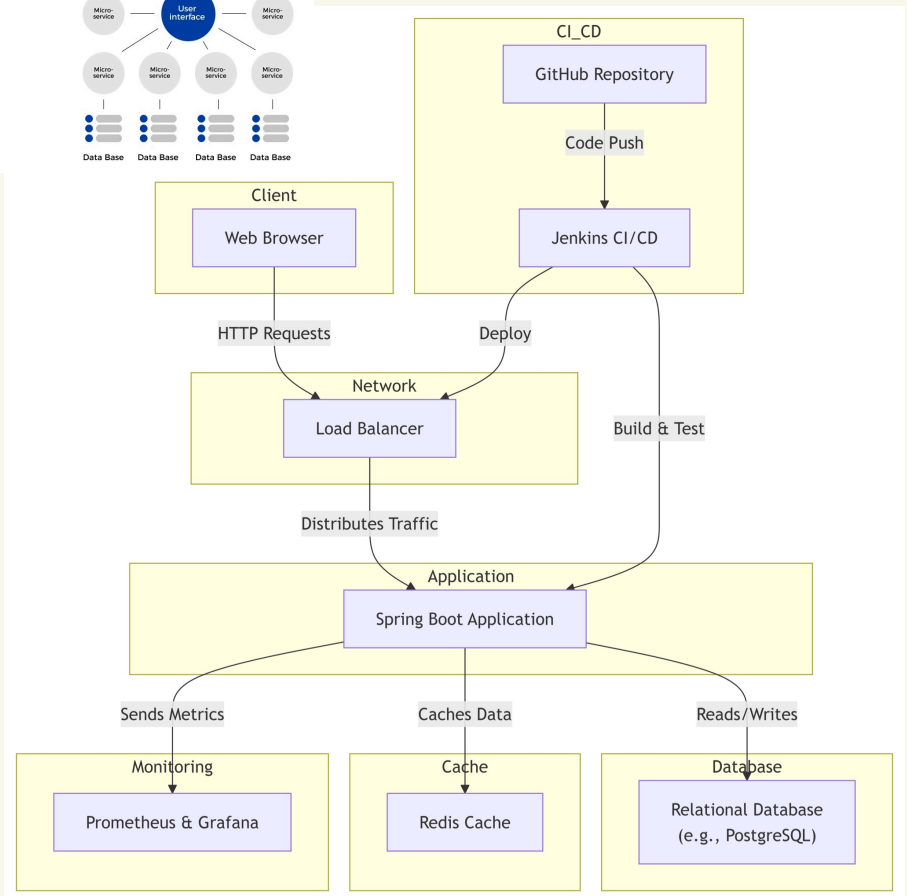
Key Differences

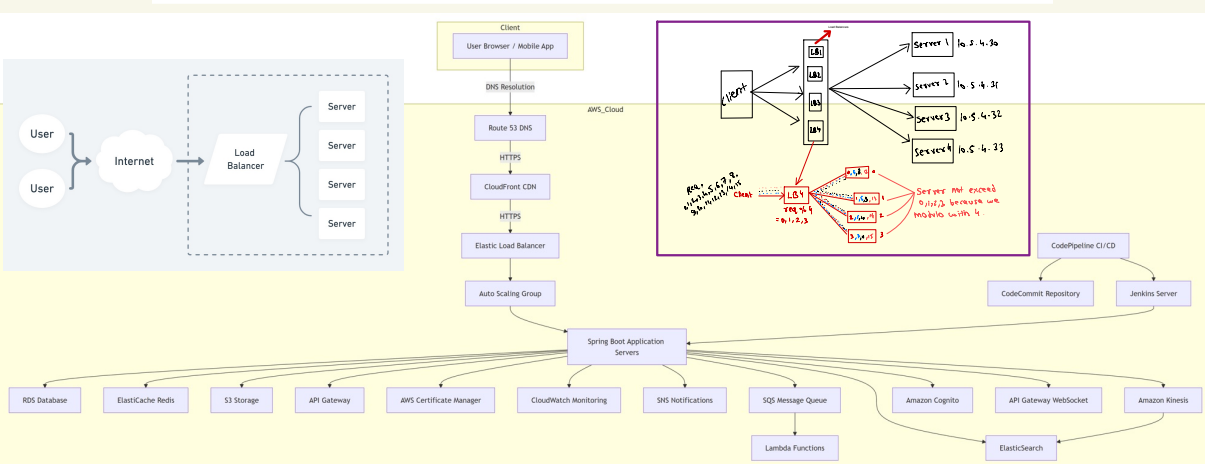
Feature	Virtual Machines (VMs)	Containers
Isolation	Stronger (each has its own OS)	Weaker (shares host OS kernel)
Startup Time	Slow (OS boot time)	Fast (lightweight processes)
Resource Usage	High (entire OS per VM)	Low (shared OS kernel)
Portability	Limited to hypervisor compatibility	Highly portable (works anywhere Docker is installed)
Use Case Example	Running legacy apps requiring different OSs	Running microservices, CI/CD pipelines

This diagram illustrates how Docker simplifies the process of application deployment by using images, containers, and registries.

Architecture Spring Boot - Microservices







1. Example System: E-commerce Platform

Microservices

1. **User Service**
   - Handles user registration, login, and profile management.
   - Provides APIs to fetch user details.
   - Example API:
     - POST /users/register
     - GET /users/{id}
2. **Product Service**
   - Manages product catalog, pricing, and inventory.
   - Provides APIs for adding, updating, and retrieving products.
   - Example API:
     - GET /products
     - POST /products
3. **Order Service**
   - Handles the creation and management of customer orders.
   - Communicates with Product and User services to validate and process orders.
   - Example API:
     - POST /orders
     - GET /orders/{orderId}
4. **Payment Service**
   - Manages payment processing and transaction history.
   - Integrates with external payment gateways.
   - Example API:
     - POST /payments
     - GET /payments/{orderId}
5. **Notification Service**
   - Sends notifications to customers (email, SMS).
   - Used by other services to notify users about events like order confirmation.
   - Example API:
     - POST /notifications

2. Core Concepts in Microservices

Key Characteristics

- **Independently Deployable**: Each service can be deployed independently.
- **Loose Coupling**: Services interact with each other through APIs.
- **Scalability**: Individual services can scale based on their load.
- **Resilience**: Failures in one service do not bring down the entire system.

Spring Boot Features for Microservices

1. **Spring Boot Starter**: Simplifies creating standalone, production-ready applications.
2. **Spring Cloud**: Adds tools for microservice architecture, like service discovery and configuration management.
3. **Spring Data**: Simplifies data access with JPA, MongoDB, etc.

3. Architecture Example

1. **Service Discovery**: Using **Eureka Server** for service registration and lookup.
2. **API Gateway**: Using **Spring Cloud Gateway** for routing and centralized API management.
3. **Load Balancing**: Using **Ribbon** or **Spring Cloud LoadBalancer**.
4. **Circuit Breaker**: Using **Resilience4j** or **Hystrix** for fault tolerance.
5. **Centralized Configuration**: Using **Spring Cloud Config Server**.
6. **Distributed Tracing**: Using **Sleuth** and **Zipkin** for debugging and monitoring.

4. Example Workflow

Use Case: Placing an Order

1. A user initiates an order from the frontend (React/Angular).
2. **API Gateway** routes the request to the **Order Service**.
3. The **Order Service**:
   - Fetches product details from the **Product Service**.
   - Validates user details from the **User Service**.
4. **Payment Service** processes the payment.
5. **Notification Service** sends an order confirmation email.

5. Implementation Steps

1. Create a Spring Boot Microservice

Example: **User Service**

java Copy

```java
@RestController
@RequestMapping("/users")
public class UserController {
    @GetMapping("/{id}")
    public ResponseEntity<User> getUser(@PathVariable String id) {
        User user = userService.findById(id);
        return ResponseEntity.ok(user);
    }

    @PostMapping("/register")
    public ResponseEntity<User> registerUser(@RequestBody User user) {
        User createdUser = userService.save(user);
        return ResponseEntity.status(HttpStatus.CREATED).body(createdUser);
    }
}
```

2. Enable Eureka Discovery

Add dependency in `pom.xml`:

xml Copy

```xml
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

Add `@EnableEurekaClient` in the main class:

java Copy

```java
@SpringBootApplication
@EnableEurekaClient
public class UserServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(UserServiceApplication.class, args);
    }
}
```

3. Centralized Configuration

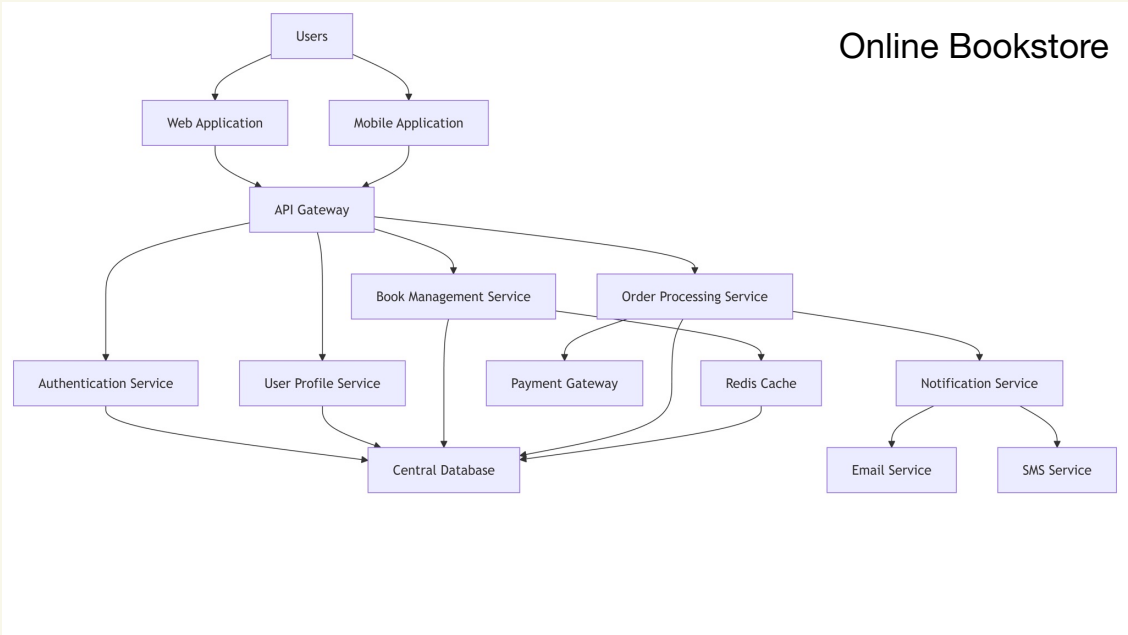
Add Spring Cloud Config dependencies and use a `config-server` for externalized properties.

6. Benefits of Microservices

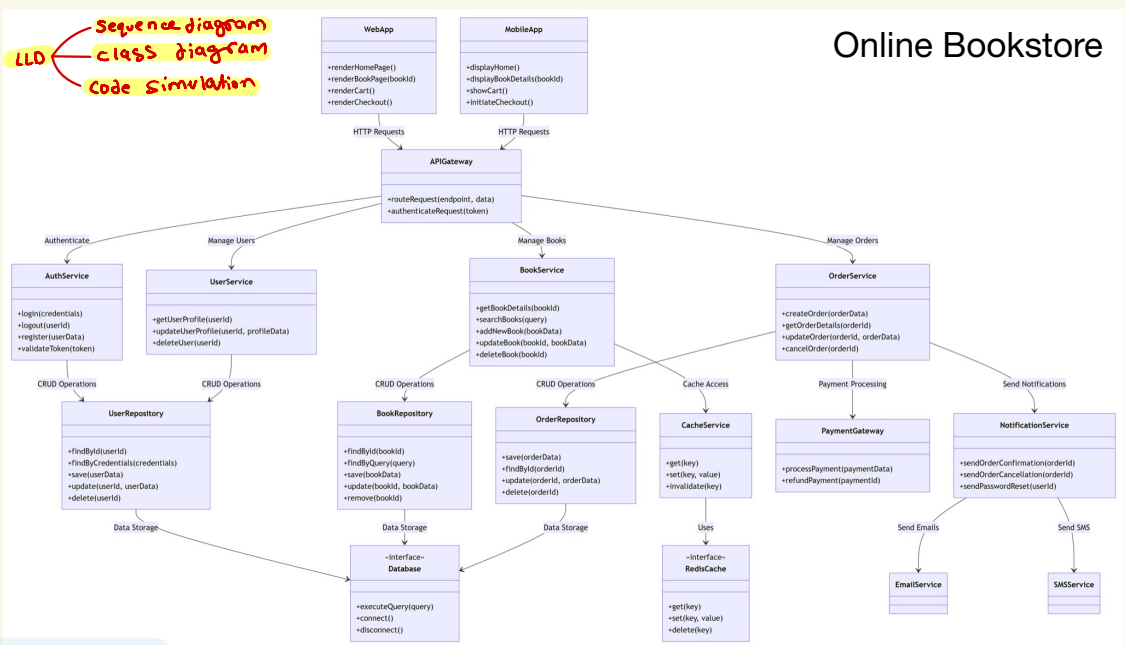
- **Scalability**: Scale services like Payment or Notification independently.
- **Flexibility**: Use different tech stacks for different services if needed.
- **Resilience**: Services can recover independently from failures.

	Monolithic Architecture	Microservices Architecture
Definition	Single, unified application where all components are interconnected.	Application divided into smaller, independent services.
Development	Easier to develop initially but becomes complex over time.	Complex development due to multiple services but scales well.
Deployment	Entire application must be deployed as a single unit.	Each service can be deployed independently.
Scalability	Limited scalability; scales as a whole.	Highly scalable; services can scale independently.
Technology Stack	Typically uses a single technology stack.	Allows different technologies for different services.
Fault Tolerance	Failure in one component can impact the entire system.	Failure is isolated to the specific service affected.
Communication	Internal calls between modules, often in-process.	Services communicate over networks using APIs (e.g., REST, gRPC).
Team Structure	Teams work on the entire application.	Teams are organized around individual services.
Testing	Easier to test as a whole but challenging for large applications.	Testing requires coordination between services.
Performance	Can be faster due to fewer network calls.	Slightly slower due to inter-service communication overhead.
Codebase Management	Single codebase; easier at first but harder as it grows.	Multiple codebases; requires good version control practices.
Maintenance	Harder to maintain as the application grows in size.	Easier to maintain as each service is smaller and self-contained.
Examples	Legacy applications, small-scale systems.	Netflix, Amazon, Uber, large-scale systems.

High Level Design



Low Level Design



What is Kafka?

- Kafka is an **open-source distributed event streaming platform** used for building real-time data pipelines and streaming applications.
- It handles large volumes of data by publishing and subscribing to streams of records.

Common Commands : Search online

Key Concepts

1. **Topic**:
   - Logical channel where messages are published.
   - Topics are **partitioned** for scalability.
2. **Producer**:
   - Sends messages to Kafka topics.
   - Writes are **append-only** to partitions.
3. **Consumer**:
   - Reads messages from topics.
   - Can be grouped in **consumer groups** to share the load.
4. **Broker**:
   - A Kafka server that stores and serves data to clients.
   - A Kafka cluster consists of multiple brokers.
5. **Partition**:
   - A topic is divided into partitions.
   - Each partition is an ordered sequence of records.
6. **Offset**:
   - A unique ID assigned to each record within a partition.
7. **Replication**:
   - Partitions are replicated across brokers to ensure fault tolerance.

Core Components

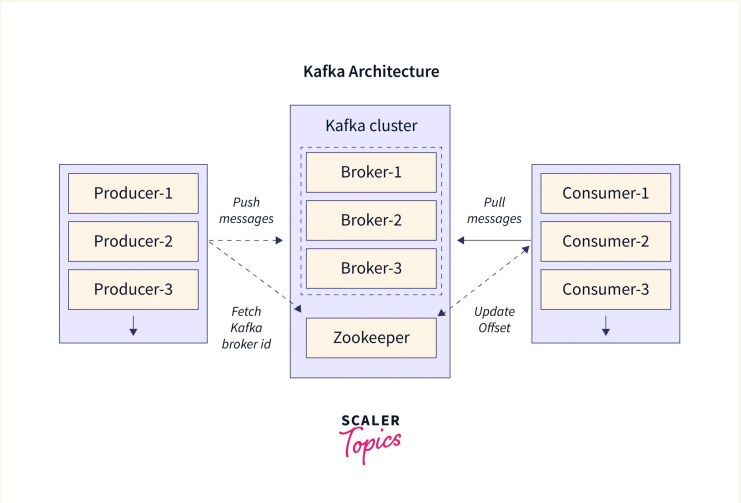
1. **Kafka Cluster**:
   - Composed of multiple brokers.
   - Managed using **ZooKeeper** (legacy) or **KRaft** (new).
2. **Producers** and **Consumers**:
   - Write and read messages using the Kafka APIs.
3. **Streams API**:
   - Processes data in real time.
4. **Connect API**:
   - Integrates Kafka with external systems (e.g., databases).

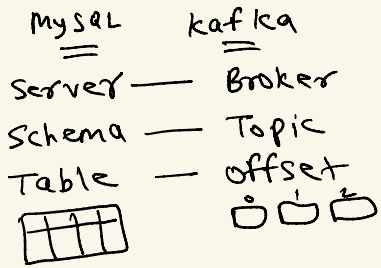
Advantages

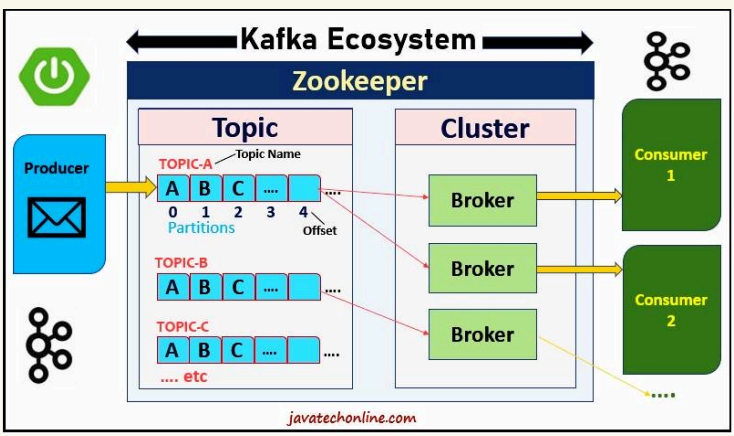
- **Scalable**: Handles large data volumes.
- **High Throughput**: Processes millions of messages per second.
- **Fault-Tolerant**: Ensures data availability via replication.
- **Durable**: Persisted messages can be retained as required.

Use Cases

- Real-time event processing.
- Log aggregation and monitoring.
- Stream processing.
- Messaging systems.







Aspect	MySQL	Kafka
Core Role	Database server for storing structured data	Distributed event streaming platform
Key Component	Server	Broker
Data Model	Tables (rows and columns)	Topics (partitions of events)
Operations	CRUD (Create, Read, Update, Delete) operations on data	Produce (write) and Consume (read) messages
Data Persistence	Stores data permanently in tables	Retains messages temporarily based on retention policy
Client Interaction	SQL queries from applications	Producers and Consumers via Kafka APIs
Data Structure	Relational data	Event-driven messages
Scaling	Vertical scaling (primarily)	Horizontal scaling (distributed cluster)
Transactions	ACID compliance	Optional, transactional messages support
Primary Use Case	Storing and querying structured data	Real-time data streaming and processing
Example Use	User data in a web app	Event-driven microservices communication

Example Scenario: Todo App

1. **TaskService (Producer)**:
   - User creates a new task via TaskService.
   - TaskService publishes a "Task Created" event to the Kafka topic `task-events`.
2. **Kafka Broker**:
   - Stores the "Task Created" message in the topic `task-events` with partitioning for scalability.
   - Messages remain available for consumers until retention policies are met.
3. **NotificationService (Consumer)**:
   - NotificationService subscribes to the `task-events` topic.
   - When it reads the "Task Created" event, it triggers an email or push notification to the user.