Spring

- **Introduction to Maven (Java Build Tool)**

What is Maven?

Maven is a build automation and project management tool for Java-based projects, developed by Apache.

Why Do We Use Maven?

1. **Build Automation**
   ☞ Automates compiling, testing, packaging (JAR/WAR), and deployment.
2. **Dependency Management**
   ☞ Automatically downloads required libraries (JARs) from **Maven Central**.
3. **Standard Project Structure**
   ☞ Follows a consistent and predefined folder layout.
4. **Plugin Support**
   ☞ Extends build functionality (e.g., for code analysis, deployment, reporting).
5. **Easy Project Management**
   ☞ Manages project versions, builds, and documentation through a single pom.xml.
6. **Integration with IDEs and CI/CD**
   ☞ Works well with tools like Eclipse, IntelliJ, Jenkins, GitHub Actions.

What is the work of Maven?

Maven automates and manages the following key tasks in a Java project:

1. **Download Dependencies**
   - Automatically downloads required .jar files from **central or remote repositories** into your **local repository**.
2. **Compile Source Code**
   - Compiles Java files in the src/main/java directory.
3. **Run Tests**
   - Executes unit tests located in src/test/java.
4. **Package the Code**
   - Packages the compiled code into .jar or .war files using the package goal.
5. **Install/Deploy the Package**
   - Installs the packaged file to the **local Maven repository** using install, or deploys it to a **remote repository** using deploy.

What is Maven Repository?

Maven Repository is a storage location from which Maven downloads project dependencies (like JAR files, plugins, etc.).

Types of Maven Repositories –

Local Repository

- Maven stores downloaded dependencies here for future use.
- Local repository location.
  Linux: /home/username/.m2
  Windows: C:/Users/username/.m2

Central Repository
- If the dependency is not in local, Maven downloads it from here.
  Provided by Maven community.
  https://mvnrepository.com/repos/central

Remote Repository
- Used for custom/internal dependencies not available in central**.**
  -Hosted on a web-server to maintain organization specific dependencies.
  -Similar to central repository.
  -Need to configure in Maven pom.xml or settings.

Maven Workflow Example:
You write a dependency in pom.xml →
Maven checks it in **Local Repository** →
If not found, downloads from **Central/Remote Repository** →
Saves it to **Local Repository** for future builds.

What is pom.xml?
-pom.xml stands for **Project Object Model.**
-pom.xml is an XML file that contains project details, dependencies, plugins, and build configurations used by Maven
-It is located into root of Maven project.
-pom.xml holds build details of the project.
-dependencies
  Third-party jars to be added into the project.
  Dependency is uniquely identified by the groupId, artifactId and version.
  All jars auto-downloaded from Maven repository and added into project CLASSPATH.
-profiles
  Maven Profiles allow you to **build the same project in different ways**, depending on the environment — like **development (dev)**, **testing (test)**, or **production (prod)**.
  Allows switching between configurations (dev/test/prod)
-build plugins
  Build Plugins allow you to customize and extend the Maven build process by adding your own tasks.
  Adds or customizes steps during build (e.g., compile, test, package, deploy)

Maven build process
Maven follows a predefined build lifecycle that defines the order of steps it performs while building a project.

Build phases :-

Build phases

- A build life cycle is divided into sequence of multiple build phases.
- Important build phases in default build life cycle.
  - validate: Check project pom.xml syntax. Downloads all dependencies (if not present in local repository).
  - compile: Compile source code of the project.
  - test: Execute given unit tests against the compiled source code using a suitable unit testing framework.
  - package: Pack the generated files into given package (jar or war).
  - install: Copy the package into the local repository. It can be used in other projects on local machine.
  - deploy: Copy the final package to the remote repository for sharing with other developers and projects.

Parent POM :-
A **Parent POM** is a pom.xml file from which other Maven projects (child modules) can **inherit configuration**.
Why use a Parent POM?
- To **avoid repetition** in multiple projects
- To manage **common dependencies**, plugins, properties

- For **centralized build configuration**
- **What gets inherited?**

Inherited from Parent Can child override?
dependencies
properties
build section
plugins
repositories

Effective POM:
The **Effective POM** is the **final merged version** of parent + child POM.
You can view it using:
mvn help:effective-pom

Spring Framework Overview
Spring :
Developed By: *Rod Johnson*
Initial Release: *2003*
Initial Goal: *Simplify enterprise Java development by replacing EJB (Enterprise JavaBeans)*

Evolution of Spring

Version	Key Highlights
Spring 1.x	XML-based configuration, Core Container, BeanFactory, early DI support
Spring 2.x	Introduced ApplicationContext, simplified AOP, support for annotations started
Spring 3.0	Full annotation support (Java 5+), became **very popular**
Spring 4.x	Java 8 features, REST support, WebSocket support
Spring 5.x	Core Spring with **Reactive programming (WebFlux)**, Java 8+, JDK 9 support
Spring 6.x	Native support for **Jakarta EE**, **AOT compilation**, **Spring Boot 3** ready

Need of Spring

Why Spring ?
-Java is Simple.
-Language syntax is easier than C, C++, ...
-Built in classes for collections, file IO, ...
-Java development includes Java coding, Unit testing, Integration testing, Configurations, ...
-Spring is lightweight comprehensive framework that simplifies Java development.

What Makes Spring Powerfull ?
-"light-weight" - basic version of Spring framework is around 2 MB.
-"comprehensive" - dependency injection
-"Simplifies Java development" - Ready-made support/wrappers for different Java technologies, Unit testing, ...

Key Principle
-Good programming practices based on interfaces and POJOs.
-Inversion of Control - Dependency injection.

Object Intialization

*Manual Initialization (Tightly Coupled)
OOP -- Object Oriented Programming -- Composition
-Outer object --> Inner object(s)

```
// ...
Car c = new Car();
c.setEngine(e);
c.setWheels(w);
c.setChasis(ch);
// ...
c.drive();
```

*Spring DI (Loosely Coupled)
Spring automates object creation as well as initialization.

```
Car c = ctx.getBean(Car.class);
c.drive();
```

This line is typically used in a **Spring application** to **get a bean from the Spring container** and call a method on it.

Spring Containers

- Manages the lifecycle of Spring beans

- Applet container -- JRE in browser to execute Applet life cycle.
- Web conatiner -- Component of web-server to execute Servlet/JSP life cycle.
- EJB container -- Component of application-server to execute EJB life cycle.
- Spring container -- Part of Spring framework to execute/manage Spring bean life cycle.
- Spring bean -- Simple Java objects created by Spring.

Spring container is also called as IoC container

Spring Advantages

- Test driven approach: Easy testing support.
  Unit testing/Mocking support in out-of-box.
  Beans can be tested independently.

- Extensive but modular and flexible.
  Extensive: Huge -- so many modules, sub-projects, wide spectrum
  Modular: Separate sub-projects/modules are available
  Flexible: Can use only modules you need on top of Spring core

- Smooth integration with existing technologies like JDBC, JNDI, ORM, Mailing, etc.
-

Boilerplate Elimination

- Eliminate boiler-plate code - ORM, JDBC, Security, etc.

  - JDBC

Load & register class (managed by spring)
Create connection (managed by spring)
Create statement (managed by spring)
Execute the query (managed by spring)
Process the result (ResultSet) -- supply SQL statement, parameters and process result.
Close all (managed by spring)

  - Hibernate

Hibernate configuration (.cfg.xml or coding) (managed by spring)

Create SessionFactory (managed by spring)
Create session (managed by spring)
transaction management (managed by spring)
CRUD operations or Query execution -- user-defined
Cleanup (managed by spring)

Unified Transaction Management

Via -- @Transactional

- Unified transaction management (local & distributed/global)

Local transactions: Within same database
Global transactions - JTA: Across the databases

Overview of Spring5 Modules

- Core Container - spring-core, spring-beans, spring-context, spring-expression
- Aop - spring-aop
- Data Access - spring-jdbc, spring-tx, spring-orm, spring-oxm, spring-data
- Web Servlet - spring-web, spring-webmvc
- Web (Reactive) - spring-webflux
- Testing – Spring-test

Spring - Core feature

- Spring IoC container
- Dependency injection

Spring IoC container :-

Inversion-of-Control

Inversion of Control is a design principle where the control of object creation and dependency management is transferred from the program (developer) to a container or framework.
Spring uses **IoC** to manage objects, inject dependencies, and build loosely coupled systems.

Traditional Approach

In a traditional application, the **dependent object** (e.g., `OrderServiceImpl`) is responsible for **creating** and **managing** its dependencies (e.g., `PaymentService`).
This creates a tight coupling between the dependent and dependency objects, making the code less flexible, harder to test, and more difficult to maintain.

Inversion of Control

With **IoC**, the **control** of creating and injecting dependencies is inverted. Instead of the dependent class creating its dependencies, an external framework (e.g., Spring) is responsible for:

1. **Creating the dependencies** (e.g., `CreditCardServiceImpl`).
2. **Injecting them** into the dependent object (e.g., `OrderServiceImpl`).

This allows the developer to focus on **business logic** instead of managing the dependencies.

IoC ensures that the framework manages the initialization and wiring of dependencies, so you can focus on implementing business methods and logic.

Core Concept of IoC

The key idea of IoC is to **delegate the control** of creating and managing dependencies to an external framework or container (e.g., Spring IoC container). This enables:

1. **Loose coupling**: Dependencies are no longer tightly coupled to the dependent class.
2. **Better testability**: Dependencies can easily be mocked or replaced for testing.
3. **Flexibility**: Dependencies can be swapped or reconfigured without modifying the dependent class.

Dependency injection :-

Dependency Injection is a **design pattern** where an object receives its dependencies from an external source (like a framework) rather than creating them itself.

Why DI?

- Promotes **loose coupling**
- Improves **testability**
- Simplifies **maintenance**
- Follows **best OOP principles**

With DI, the framework injects the dependencies into your dependent object via:

1. **Constructor**
2. **Setter**
3. **Field (via annotations like @Autowired)**

- OOP (Bottom-up) vs POP (Top-down)

Feature	**OOP (Bottom-up)**	**POP (Top-down)**
Style	Objects first, then logic is formed	Logic first, then functions & modules created
Approach	Build from components → system	Plan overall flow → break into functions
Dependency creation	Objects control their own dependencies (tight coupling)	Functions call other functions directly (procedural)
Example	Car creates its own Engine object	A top-level function creates and manages everything
Control Flow	Objects collaborate	Functions follow a fixed sequence

IoC fits more naturally into OOP, where object responsibilities can be handed off to a container

- Manual Object Initialization vs DI Object Initialization (Inverted)

  - You **explicitly create** objects and manage their dependencies:

```
Engine e = new Engine();
Car c = new Car(e);
```

You control **when and how** objects are created.
Tightly coupled — Changing Engine needs changes in Car.

  - DI **(Dependency Injection)** Initialization **— IoC Style**

Instead of your code creating dependencies, the framework injects them.
You **declare dependencies**, and a **container (like Spring)** injects them:
Field Injection:

```
@Component
public class Car {
@Autowired
private Engine engine;
}
```

OR constructor injection:

```
public class Car {
private Engine engine;

@Autowired
public Car(Engine engine) {
  this.engine = engine;
    }
}
```

Now **Spring controls object creation** — not you.
Spring container manages object creation and injection.
Loose coupling, better maintainability and testing.

Spring Beans :-

- Spring beans are Java class objects instantiated by Spring container.
- Beans are created and initialized as per user-defined configuration (bean definition).
- Spring bean classes are simple Java POJO classes with one or more business logic method.

Spring configuration :-

- Spring allows configuring beans in different ways.
  1. XML config
  2. Java config
  3. Mixed config

XML config

- Early versions of Spring (before Spring 2.5) support only XML configuration.
- Spring beans are declared into Spring bean configuration file.
- These files follow fixed syntax/grammer (in terms of XSD).
- XML configuration allows various config
- Initializing properties (primitive types)
- Initializing dependency beans
- Initializing collections (list, maps, etc)
- Defining bean initialization and de-initialization
- Defining bean scopes etc.

Typically ClassPathXmlApplicationContext reads the bean config file and create Spring beans at runtime.

Java config

- Supported from Spring 2.5 onwards and makes extensive use of annotations.
- The bean creation is encapsulated in @Configuration class's @Bean methods.
- Java config also allows defining beans, their dependencies, scopes, initialization, etc.
- An application may have multiple configuration classes. A @Configuration class may import another using @Import.
- Spring Boot recommends java config. In Spring Boot, by default it detects all @Configuration classes in the main package.

Typically AnnotationConfigApplicationContext gets config from @Configuration classes and create Spring beans at runtime

Mixed config

- XML configuration can process annotations like @Autowired, @Qualifier, @PostConstruct and @PreDestroy using <context:annotation-driven/>. It can
- also detect stereo-type annotations using <context:component-scan>.
- In Java config, @Configuration class may import XML config using @ImportResource.
- When a class is marked with **stereotype annotation**, Spring **auto-detects it during startup**, **creates a bean object**, and **manages it in the IoC container**.

-

What is a Stereotype Annotation in Spring?

A **stereotype annotation** is a special annotation used in Spring to **mark a class as a Spring-managed bean** — so that Spring can **automatically detect and register** it in the IoC container during startup.

Stereotype annotations are used to mark classes as Spring beans with specific roles (like service, controller, repository).
They help Spring automatically detect and register those beans during the startup process.

@Component, @Service, @Repository, @Controller, @RestController This are the Stereotype Annotations

Stereo-type annotations

- Spring container can auto-detect certain beans (without defining as @Bean method or XML config). It also manage their bean life cycle.
- Following annotations are auto-detected by Spring container using @ComponentScan.
    - @Component -- general-purpose (no special significance) spring bean.

- @Service -- spring beans containing business logic
- @Repository -- spring beans handling data/database connectivity
- @Controller -- spring beans handling navigation, user-interaction in Spring web-mvc applications.
- @RestController -- It is @Controller used for REST services.
- @Configuration -- To be used for spring annotation config (not to be used as spring bean).

Stereo-type annotations are always written at class level.
Using @ComponentScan, all classes with above annotations in given package and its sub-packages will be instatiated as Spring beans.
If no package is given, all classes in current package will be scanned for stereo-type annotations.
This behaviour can be customized using includeFilters and/or excludeFilters.
XML configuration equivalent of @ComponentScan `<context:component-scan>`.

ApplicationContext

- Spring container is created while ApplicationContext is created.
- ApplicationContext enable accessing Spring container features in the application.
- ApplicationContext is an interface and have various implementations for different scenarios/applications.
- ApplicationContext
    - ClassPathXmlApplicationContext
        - ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml"); // beans.xml in resources or src dir.
    - FileSystemXmlApplicationContext
        - FileSystemXmlApplicationContext ctx = new FileSystemXmlApplicationContext("/home/nilesh/beans.xml"); // beans.xml is in some folder.
    - AnnotationConfigApplicationContext
        - AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class); // AppConfig is @Configuration class
    - WebApplicationContext (interface)
        - XmlWebApplicationContext
        - AnnotationConfigWebApplicationContext
- ApplicationContext (Spring Container) create all (singleton) beans when application context is loaded.
- ApplicationContext interface itself is inherited from BeanFactory interface.
    - It provides basic facility of Beans loading and initialization.
- Typical spring applications have single ApplicationContext.

Dependency Injection

- https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-factory-collaborators
- Object dependencies are typically defined as
  - Constructor arguments
  - Property (Setter) arguments
  - Factory method arguments
- The spring container inject them into beans immediately after bean creation.
- Advantages of Dependency Injection
  - Cleaner code
  - Loose coupling
  - Location transparency (of dependency)
  - Easier testing/mocking
- There are two major varients of dependency injection.
  - Constructor based DI
  - Setter based DI

Dependency Resolution Process

- The ApplicationContext is created and initialized with configuration metadata (XML or annotations) that describes all the beans.
- For each bean, its dependencies are expressed in the form of properties or constructor arguments. These dependencies are injected when the bean is actually created.
- Each property or constructor argument can be a value (primitive type) or reference of another spring bean.
- The value is converted from its specified format to the actual type of that property or constructor argument.

Annotation Based configuration

- @SpringBootApplication = @ComponentScan + @Configuration + @EnableAutoConfiguration

- @ComponentScan
  - Auto detection of spring stereo-type annotated beans e.g. @Component, @Service, @Repository, @Configuration, @Controller, @RestController, ...
  - By default basePackage is to search into current package (and its sub-packages).

- @ComponentScan("other.package") can be added explicitly to search of beans/config into given package.

- @Configuration

  - Spring annotation configuration to create beans.
  - Contains @Bean methods which create and return beans.
  - @Configuration classes are also auto-detected by @ComponentScan

- @EnableAutoConfiguration annotation
  - Intelligent and automatic configuration
  - Auto-configuration class are internally Spring @Configuration classes.
  - @Conditional beans
    1. @ConditionalOnClass
    2. @ConditionalOnMissingBean

  - **Spring Boot specific annotation** that tells Spring:
    1. "Try to automatically configure beans based on the classpath dependencies and application.properties."
    2. Internally uses @Import(AutoConfigurationImportSelector.class) to load configurations dynamically.

Spring Core Container

Spring Core Container is the foundation of the Spring Framework. It is responsible for managing the complete lifecycle of beans — creation, wiring, configuration, and destruction.

- BeanFactory

- BeanFactory is the root interface for the Spring IoC (Inversion of Control) container. i.e Spring container (ApplicationContext) extends BeanFactory.
- It manages beans, i.e., it creates, configures, and wires beans.

Why do we use BeanFactory :-

Purpose	Explanation
Lightweight container	Uses less memory; good for **simple** or **mobile** apps.
Lazy initialization	Beans are created **only when needed**, not at startup.
IoC and Dependency Injection (DI)	Handles dependency management and injection.
Manual control	Gives developers more control compared to ApplicationContext.

When to Use BeanFactory?
Use BeanFactory when:

You want a lightweight application (e.g., mobile or embedded systems).
You need lazy loading of beans.
You don't need advanced features like event handling, internationalization, etc.

BeanFactory vs ApplicationContext

Feature	BeanFactory	ApplicationContext
Lazy bean loading	Yes	No (eager by default)
Advanced features	No	Yes (events, AOP, etc.)
Common in Spring Boot	Rare	Always used

- What is ApplicationContext?

  - ApplicationContext is the central interface for the Spring IoC container.
    It is responsible for:
  - Creating, configuring, and managing beans
  - Handling dependency injection
  - Supporting advanced features like event handling, AOP, and internationalization

  It extends BeanFactory, so it includes all BeanFactory features plus many more.

  Features of ApplicationContext:

Feature	Description
Eager bean loading	Beans are created at startup (by default)
Internationalization	Supports MessageSource for i18n messages
Event publishing	Can publish/subscribe to events (e.g., ContextRefreshedEvent)
Bean lifecycle callbacks	Supports @PostConstruct, @PreDestroy, InitializingBean, DisposableBean
Annotation scanning	Detects beans via @Component, @Service, etc.
Environment abstraction	Supports property files (application.properties, etc.)

What is Eager Bean Loading in Spring?
Eager loading means:
Spring creates all singleton beans when the application starts, not when they're first used.
What Does "Beans are created at startup by default" Mean?

- When you run a Spring application (using ApplicationContext or Spring Boot),
- Spring goes through all classes annotated with @Component, @Service, @Repository, etc.

- It immediately creates those beans and puts them in the **Spring container (IoC)**.

Even if you **never use** that bean in your code — it's **still created** during startup.
Lazy Loading (Opposite of Eager):

- Spring does not create the bean until it's requested/used for the first time.

Common Implementations:

Class	Description
ClassPathXmlApplicationContext	Loads beans from XML in classpath
FileSystemXmlApplicationContext	Loads beans from XML in file system
AnnotationConfigApplicationContext	Loads beans from Java @Configuration class
WebApplicationContext	Special context for Spring MVC (Web apps)

- Bean

  What is a Bean in Spring?
  In Spring Framework, a bean is:
  " A Java object that is managed by the Spring container. **"**
  When you say Spring bean, you're referring to:
  An object that is created, configured, and maintained by Spring.

  Why Use Beans?
  Because Spring handles:
  - Object creation
  - Dependency injection
  - Lifecycle management
  - Wiring between objects

  So you don't need to create and connect objects manually.

  How Are Beans Created?
  -There are 3 main ways to define beans in Spring:
  -Using Annotations (Common in Spring Boot):
  -Using @Bean inside @Configuration class:
  -Using XML Configuration (Old style):

  Default Scope of a Bean

Scope	Description
singleton	One object per Spring container (default)
prototype	New object every time you request it
request	One per HTTP request (Web only)
session	One per HTTP session (Web only)

- @Component

  - **The @Component annotation is used to:**
  - Mark a Java class as a Spring-managed bean so that the Spring IoC container c an automatically detect and register it during classpath scanning.
  - **Why use @Component?**
  - It eliminates the need to define the bean manually in XML or Java config.
  - Enables automatic detection when used with @ComponentScan.

  Example:

```java
@Component
public class Car {
    public void drive() {
        System.out.println("Driving...");
    }
}
```

Spring will:

Detect this class during @ComponentScan

Create a **singleton object** (bean) of Car

Manage its lifecycle and dependencies

- @Service / @Repository / @Controller

Specializations of @Component:

Annotation	**Purpose**
@Service	For business/service layer
@Repository	For DAO/persistence layer
@Controller	For Spring MVC web controllers

- @Configuration
  Marks a class as **Java-based configuration** for Spring.
  Contains @Bean methods.
  Spring treats it like an XML <beans> file.

- @Bean
  Used inside @Configuration class to **define a bean manually**.
  Return value is registered as a bean.

- @ComponentScan
  Tells Spring where to **look for beans** (stereotype-annotated classes).

- @Autowired
  Tells Spring to **inject** a dependency automatically.
  Can be applied to constructor, field, or setter.

- @Qualifier
  Used with @Autowired to resolve **ambiguity** when **multiple beans** of the same type exist.

- Bean Lifecycle (3 Phases)

Phase	**Description**
Init	Bean is created and dependencies injected
Use	Bean is used in application
Destroy	Cleanup before Spring container closes

Can use:
@PostConstruct, @PreDestroy
InitializingBean, DisposableBean

- IoC (Inversion of Control)
  Spring container takes over:
  Object creation
  Dependency injection
  Lifecycle management

- Dependency Injection

  Spring container inject dependency object after bean object creation.
  There are two major varients of dependency injection.
  - Constructor based DI
  - Setter based DI
  - Factory Method-Based DI

  Setter based DI :- Setter-based Dependency Injection is a method in Spring where **dependencies are injected into a bean using setter methods**.
  Example :

```
@Component
public class Car {
    private Engine engine;

    @Autowired
    public void setEngine(Engine engine) {
        this.engine = engine;
    }

    public void drive() {
        engine.start();
    }
}
```

  Engine is a dependency.
  Spring injects it using the setEngine() method.

  Setter Injection uses public **setter methods** to inject dependencies.
  Uses @Autowired on setter methods (or @Inject from javax).
  Best for **optional or changeable dependencies**.
  Can be combined with @Qualifier if multiple beans are present.

  Constructor based DI :-
  Spring injects the required dependencies **through a class constructor** at the time of **bean creation**.

  Example :

```
@Component
public class Car {
    private Engine engine;

    @Autowired  // Optional if only one constructor
    public Car(Engine engine) {
```

```
        this.engine = engine;
    }

    public void drive() {
        engine.start();
    }
}
```

Spring will:
Automatically detect the constructor
Create an Engine bean
Inject it into the Car bean at startup

Constructor vs Setter Dependency Injection — Core Differences

Criteria	Constructor Injection	Setter Injection
Injection Time	At the time of **bean creation**	Immediately **after bean creation**
Dependency Requirement	Enforces **mandatory** dependencies	Allows **optional** dependencies
Immutability	Promotes immutability (final fields possible)	Mutable – can change the dependency after creation
Code Clarity	Clear dependencies in one place (constructor)	Spread out (setters)
Testability	Easier to write tests — dependencies in constructor	Slightly harder — might require additional setup
Use Case	Use when **all dependencies are required**	Use when **some dependencies are optional**
@Autowired usage	Optional (Spring 4.3+ if single constructor)	Required unless using field injection

Constructor DI is best when you want to **enforce required dependencies** — Spring **won't create the bean** if it can't satisfy the constructor arguments.

Setter DI is best when the dependency is **optional** or when the bean **needs to be reconfigured** post-instantiation.

Creating Spring Java SE application in STS

1. Create New Project

- Open STS → File → New → Spring Project
- Choose Simple Spring Project or Spring Starter Project (without web dependency)
- Give a name (e.g., spring-core-demo)
- Use Java 8+ and Maven

2.Add Spring Core Dependency (pom.xml)

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.3.30</version> <!-- or latest -->
    </dependency>
```

</dependencies>

3.Create a Java Config Class

```
@Configuration
@ComponentScan("com.example")
public class AppConfig {
}
```

4.**Create a Component Class (Bean)**

```
@Component
public class Car {
    public void drive() {
        System.out.println("Car is driving...");
    }
}
```

5.Main Class (to run)

```
public class MainApp {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new
    AnnotationConfigApplicationContext(AppConfig.class);

        Car car = context.getBean(Car.class);
        car.drive();

        context.close();
    }
}
```

Scopes of spring beans

In Spring, bean scope defines how many instances of a bean will be created and how long they will live within the container.

Spring Bean Scopes

- Using a bean definition (@Bean or Stereo-type annotations + Other config), one or more bean objects can be created.
- The bean scope can be set in XML or annotation.
  - `<bean id="___" class="___" scope="singleton|prototype|request|session" />`
  - @Scope("singleton|prototype|request|session") on @Bean / @Component ...
- Spring 5 supports six different bean scopes. Four scopes are valid only if you use a web-aware ApplicationContext.

Scope	Description
singleton	(Default) A single object instance for each Spring IoC container.
prototype	Any number of object instances (new instance for each access).
request	A single object for current HTTP request.
session	A single object for current HTTP session.
application	A single object for current application i.e. ServletContext.
websocket	A single object for current websocket.

Singleton

- Single bean object is created and accessed throughout the application.
- BeanFactory creates object when getBean() is called for first time for that bean.
- All singleton bean objects are created when ApplicationContext is created.
- For each sub-sequent call to getBean() returns same object reference.

- Reference of all singleton beans is managed by spring container.
- During shutdown, all singleton beans are destroyed (@PreDestroy will be called).

Singleton class vs Singleton bean :-

- Design patterns are solutions to the well-known problems.
- Singleton is a design pattern. It enable access to a single object throughout the application.
- In OOP languages (like Java), Singleton class is created so that only one object of the class is available
- for the use in the whole application.
- In Spring, singleton is scope of spring bean. Same bean object (identified by the same id) will be
- accessible in the whole application. However spring bean class is a simple Java class and hence multiple
- objects of that bean class possible

Prototype

- No bean is created during startup.
- Reference of bean is not maintained by ApplicationContext.
- Beans are not destroyed automatically during shutdown.
- Bean object is created each time ctx.getBean() is called.

Dependency with different scopes

Singleton bean inside prototype bean

- Single singleton bean object is created.
- Each call to getBean() create new prototype bean. But same singleton bean is autowired in them.
- e.g. OrderImpl <>--- RestaurantImpl

Prototype bean inside singleton bean

- Single singleton bean object is created.
- While auto-wiring singleton bean, prototype bean is created and is injected in singleton bean.
- Since there is single singleton bean, there is a single prototype bean.
- e.g. Outer1 <>--- Inner1

Need multiple prototype beans from singleton bean?

1. Using ApplicationContext

- The singleton bean class can be inherited from ApplicationContextAware interface or @Autowired ApplicationContext.
- This ApplicationContext can be used to create new prototype bean each time (as per requirement).
- e.g. Outer2 <>--- Inner2

2. Using @Lookup method

- The singleton bean class contains method returning prototype bean.

- If method is annotated with @Lookup, each call to the method will internally call ctx.getBean(). Hence for inner prototype beans, it returns new bean each time.
- e.g. Outer3 <>--- Inner3

Spring Beans life cycle

- The Spring Bean Lifecycle refers to the sequence of steps that a bean goes through from creation to destruction, fully managed by the Spring Container.
- Life cycle of an java object is governed by the container that creates the object.
- Container do call certain methods the object to initialize it or give some information.
- Java applets are executed by JRE plugin in the browser (referred as applet container). It used to call methods of applet.
- init(), start(), paint(), stop(), destroy().

- Java servlets are executed by web container in the Java web server. It calls the certain methods of the servlet object.
- init(), service(), destroy()
- The life cycle methods are callback methods i.e. they are implemented by the programmer and are invoked by the container.
- init() method is called by the container and all container services are accessible from there. This is not possible while construction of the object.

⚠ **Note:**
Applets are **now obsolete** in modern Java versions and browsers no longer support them due to security concerns.

Lifecycle Phases of a Spring Bean

1. **Instantiation**
   Spring container **creates the bean object** using constructor (or factory method).

2. **Populate Properties (Dependency Injection)**
   Spring **injects dependencies** via constructor, setter, or field injection.

3. **Set Bean Name**
   Bean name is set using BeanNameAware.

4. **Set Bean Factory**
   If bean implements BeanFactoryAware, Spring passes reference to the BeanFactory.

5. **Set ApplicationContext**
   If bean implements ApplicationContextAware, context reference is passed.

6. **Pre-initialization (Post Processors)**
   BeanPostProcessor.postProcessBeforeInitialization() is called.

   BeanPostProcessor :-
   To perform certain actions on mutliple beans initialization, custom BeanPostProcessor is used.
   There are multiple pre-defined BeanPostProcessor.
   AutowiredAnnotationBeanPostProcessor
   InitDestroyAnnotationBeanPostProcessor
   BeanValidationPostProcessor

7. **Initialization**
   - @PostConstruct method runs (if defined).
   - OR custom init method (declared in XML or with @Bean(initMethod="..."))
   - OR if implementing InitializingBean.afterPropertiesSet().

8. **Post-initialization (Post Processors)**
   BeanPostProcessor.postProcessAfterInitialization() is called.

9. **Bean is ready to use**
10. **Destruction**
    - @PreDestroy method runs (if defined).
    - OR custom destroy method (via XML or @Bean(destroyMethod="..."))
    - OR if bean implements DisposableBean.destroy().

Bean creation

1. Constructor
2. Fields/setters initialized (DI)
3. BeanNameAware.setBeanName()
4. ApplicationContextAware.setApplicationContext()
5. CustomBeanPostProcessor.postProcessBeforeInitialization()
6. @PostConstruct method invoked
7. InitializingBean.afterPropertiesSet()
8. CustomBeanPostProcessor.postProcessAfterInitialization()
9. Bean is ready to use

Bean destruction

1. @PreDestroy method invoked
2. DisposableBean.destroy()
3. Object.finalize()
4. Bean will be garbage collected

Auto-wiring vs Explicit Wiring in Spring

Feature	Autowiring	Explicit Wiring
Definition	Spring automatically injects dependencies.	Developer manually specifies the dependency to inject.
Configuration Style	Uses annotations like @Autowired or XML autowire="...".	Uses XML <property> or Java @Bean with method parameters.
Code Simplicity	Less boilerplate code.	More control, but more verbose.
Flexibility	Less flexible; may need @Qualifier for conflicts.	Fully flexible; developer defines everything clearly.
Error Handling	May throw errors if no matching bean is found.	Less error-prone if configured correctly.
Maintainability	Easier for small projects.	Better for large, complex projects.

When to Use What?

- Use **Autowiring** for rapid development or smaller applications.
- Use **Explicit Wiring** when you want **full control** over bean injection and dependencies (e.g., enterprise projects).

Auto-wiring using annotation configuration

- Spring provides annotation-based configuration to **automatically inject dependencies** into beans.
-

Common Anotaions For Autowiring :-
@Autowired
Injects a bean **by type**. Can be used on constructor, setter, or fields.
@Qualifier
Used with @Autowired to resolve **ambiguity** when multiple beans exist.
@Primary
Marks a bean as the **default** choice when multiple candidates are present.

Where Can Be Autowired To Be Used? :-
1.Field Injection

```
@Component
public class Car {
 @Autowired
private Engine engine;
}
```

2.Setter Injection

```
@Component
public class Car {
 private Engine engine;
  @Autowired
  public void setEngine(Engine engine) {
 this.engine = engine;
 }
}
```

3.Constructor Injection

```
@Component
public class Car {
 private final Engine engine;
@Autowired
public Car(Engine engine) {
this.engine = engine;
 }
}
```

Notes:

- If there's only one matching bean, Spring autowires automatically.
- If there are multiple beans, use @Qualifier("beanName") to specify which one.
- Constructor-based autowiring is recommended for **immutability and testing**.

Field / Setter Level Autowiring

- @Autowired directly on a field or setter method.
- Spring injects the dependency after object creation (i.e., after constructor call).
- Good for optional or mutable dependencies.
- Field injection is concise but not recommended for unit testing (no easy mocking).
- Setter injection allows flexibility and better testability.

Constructor Autowiring

- @Autowired (optional in Spring 4.3+) on a constructor.
- Dependency is injected at the time of bean creation.
- Recommended for mandatory dependencies.
- Promotes immutability and is more testable.
- Helps with cleaner code (especially in final fields).

#Introduction to Spring Boot:

Purpose:
Spring Boot simplifies Spring application development by providing opinionated defaults and auto-configuration, reducing boilerplate code and setup effort.

Auto-Configuration:
Automatically configures Spring components based on classpath dependencies and application

properties, minimizing manual XML or Java config.

Starter Dependencies:
Predefined Maven/Gradle dependencies (e.g., spring-boot-starter-web, spring-boot-starter-data-jpa) bundle commonly used libraries for specific tasks, ensuring compatibility.

Embedded Servers:
Spring Boot embeds servers like Tomcat, Jetty, or Undertow so you can run web apps as standalone Java applications (via java -jar), no need to deploy WAR files.

Spring Boot CLI:
A command-line tool allowing rapid Spring Boot app development using Groovy scripts or simplified project bootstrap.

SpringApplication Class:
The entry point launcher for Spring Boot apps. Its static run() method bootstraps the application context and triggers auto-configuration.

Opinionated Defaults:
Spring Boot chooses default configurations (like port 8080, default data sources) which can be easily overridden via application.properties or YAML.

Actuator Module:
Provides production-ready features like health checks, metrics, and monitoring endpoints for live application insights.

No XML Configuration:
Spring Boot embraces Java-based configuration and convention over configuration, minimizing or eliminating XML config files.

Auto-Configuration Ordering & Exclusions:
Auto-configuration classes are loaded based on classpath and conditions; sometimes you need to exclude specific auto-configs to avoid conflicts or customize behavior.

Profile-Specific Configurations:
Profiles (@Profile, application-{profile}.properties) let you run different configs (dev, test, prod) easily, but misconfigurations can cause silent failures.

Starter Dependency Versions:
Starters manage dependencies for you, but mixing manual versions can cause dependency conflicts or version mismatch issues.

Minimal Setup:
A basic Spring Boot app can be started with just one class annotated with @SpringBootApplication which combines:
@Configuration
@EnableAutoConfiguration@Componen
tScan

#Hello Spring Boot Java SE application & its execution:

1. Basic Structure:

A minimal Spring Boot Java SE app requires only one class with the main() method.

Entry class is annotated with:
@SpringBootApplication — a meta-annotation that includes:
@Configuration: Marks class as a config source.
@EnableAutoConfiguration: Enables Spring Boot's auto-setup.
@ComponentScan: Automatically scans the package for components.

2. The Main Method:

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

SpringApplication.run():
Starts the Spring ApplicationContext
Triggers auto-configuration
Scans for beans/components
Starts embedded server (if it's a web app)

3. Java SE vs Web:
Spring Boot works without web dependencies too.
In Java SE (non-web), SpringApplication.run() creates an ApplicationContext but no embedded server is started.

4. Executing the Application
Build a JAR using Maven/Gradle:

```
mvn clean package
java -jar target/myapp.jar
```

The application runs as a standalone Java process.

5. application.properties or .yml (Optional):
Can be used to customize settings even in Java SE apps, e.g.:

```
spring.main.banner-mode=off
```

6. @ComponentScan Scope:
It only scans current package and sub-packages.
Beans outside this scope won't be detected unless explicitly configured.

7. Silent Bean Errors:
If a required component isn't found, app may start successfully if not used immediately.
Be cautious of lazy bean initialization hiding missing beans.

8. SpringApplication.run() Returns ConfigurableApplicationContext
You can use it to manually retrieve beans in Java SE context:

```
ConfigurableApplicationContext ctx = SpringApplication.run(MyApp.class, args);
MyService service = ctx.getBean(MyService.class);
```

9. Exit Handling in Java SE
Use SpringApplication.exit() for graceful shutdown and exit codes.

10. Disabling Auto-Config for Java SE
You can exclude web-related auto-config:

@SpringBootApplication(exclude = {WebMvcAutoConfiguration.class})

#Stereo-type annotations

1. Definition
Stereotype annotations are Spring-provided annotations used to mark classes as Spring-managed components for automatic detection and registration.

Common Stereotype Annotations
@Component : Generic Spring-managed component (Generic/Any Layer)
@Service : Marks service layer class (Business Logic Layer)
@Repository : Marks data access layer (DAO) class (Persistence Layer)
@Controller : Marks web controller (MVC) (Web Layer)
@RestController : @Controller + @ResponseBody (REST APIs)

2. Component Scanning
Spring detects these annotations via component scanning, typically triggered by @ComponentScan.
Classes must reside in the same or sub-packages of the class annotated with @SpringBootApplication.

@Component is the base stereotype
All others (@Service, @Repository, etc.) are meta-annotated with @Component.
Functionally equivalent at runtime, but give semantic meaning for tooling, AOP, and exception translation.

@Repository adds exception translation
Converts JPA/Hibernate exceptions into Spring's DataAccessException hierarchy.
Silent benefit — only works if the annotation is used correctly.

@RestController = @Controller + @ResponseBody
All handler methods return data directly (typically JSON).
If you use @Controller without @ResponseBody, Spring tries to return a view name instead of raw data.

Misplaced Stereotypes Don't Error
If you annotate a class with @Service but forget to place it in a scanned package, no error is thrown, but the bean isn't registered — tricky to debug!

Overusing @Component
Though generic, overusing @Component without proper semantics reduces code readability and may confuse layered design.

Custom Stereotypes
creating your own stereotype:

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Component
```

public @interface MyCustomComponent {}

Useful for grouping behaviors or roles in large apps.

~All stereotype annotations register beans in the Spring container, but @Repository and @Service offer layer-specific behaviors like exception translation or better AOP support.

#ORM Concept

1. Definition
ORM (Object-Relational Mapping) is a technique that maps Java objects to relational database tables, allowing you to interact with the database using objects instead of raw SQL.

Core ORM Goals

Abstraction: Hides JDBC code & SQL complexities
Productivity: Save, update, delete using objects
Portability: Supports multiple DBs with minor config changes
Maintainability: Centralized domain model for data and logic

2. Core ORM Components
Entity : A class mapped to a table using @Entity
Table : A DB table represented by an entity class
Primary Key : Field annotated with @Id
Relationships : Modeled using @OneToOne, @OneToMany, etc.
Persistence Context : First-level cache holding managed entities

Dual Identity

An entity has two identities:
Java Object Identity (==)
Database Identity (Primary Key)

Transient vs. Persistent

A Java object becomes persistent once saved in DB via EntityManager or repository.
Until then, it’s transient and not managed by ORM.

Dirty Checking

ORM tools like JPA detect changes to persistent objects and auto-update them in DB at flush/commit.
Tricky part: No explicit update needed, but only if object is still in persistence context.

N+1 Query Problem

Happens when fetching an entity with lazy-loaded relationships in a loop.
Results in many unintended DB queries.
Fix: Use fetch join or @EntityGraph.

Lifecycle States (JPA)

Transient : Not managed, no DB record
Persistent : Managed by persistence context
Detached : Was persistent, now unmanaged
Removed : Marked for deletion from DB

ORM != JPA

ORM is the concept;
JPA (Java Persistence API) is a specification implementing ORM;
Hibernate is the most common JPA provider.

Object Graph Complexity

Mapping deep/nested object graphs with ORM is powerful, but can easily lead to:
Infinite recursion
Lazy-loading issues
Circular references in JSON serialization

SQL Generation

ORM generates SQL dynamically.
You must understand the generated SQL to avoid performance surprises.

Schema Auto-generation (Optional)

ORM can generate schema from entities using spring.jpa.hibernate.ddl-auto but should be disabled in production for stability.

ORM bridges the gap between object-oriented programming and relational databases, but you must understand both sides (Java & SQL) to use it effectively.

#JPA

JPA is a Java specification for ORM (Object-Relational Mapping).
It defines APIs and annotations to map Java objects (entities) to relational database tables.

JPA itself is not an implementation. Popular implementations:
Hibernate
EclipseLink
OpenJPA

2. Key Annotations
@Entity : Marks class as a persistent entity
@Id : Specifies primary key
@GeneratedValue : Auto-generates PK using strategy
@Column : Maps a field to a table column
@Table : Customizes table name
@OneToOne, @OneToMany, @ManyToOne, @ManyToMany : Defines relationships

3. Core Interfaces & Classes
EntityManager : Core interface to interact with persistence context
EntityTransaction : Manages transactions (in Java SE apps)
Persistence : Bootstrap class for SE apps
PersistenceContext : Container holding managed entities

JPA is just an API

You always use a provider (like Hibernate) behind the scenes.

Persistence Context = First-Level Cache

Every managed entity is cached in memory within a transaction scope.
Changes made are automatically detected (dirty checking).

Entity Lifecycle States

Transient : New object, not yet managed or saved
Persistent : Managed by EntityManager
Detached : Was persistent, now disconnected
Removed : Scheduled for deletion

@GeneratedValue Strategies
AUTO: Let provider decide (usually sequence/table-based)
IDENTITY: DB auto-increment column
SEQUENCE: Uses a named sequence
TABLE: Uses a table to simulate sequence (slowest)

Fetch Type Confusion
@ManyToOne and @OneToOne: Default is EAGER
@OneToMany and @ManyToMany: Default is LAZY

Lazy fetching can cause LazyInitializationException if entity is accessed outside transaction.

JPQL vs Native SQL
JPQL (Java Persistence Query Language) is object-based, e.g.:
SELECT e FROM Employee e WHERE e.name = :name

Native SQL can be used when performance/tuning is critical.

Cascading Pitfalls
CascadeType.ALL applies all actions (persist, merge, remove) — can accidentally delete child entities if not used carefully.

Transactions Are Required
All JPA operations must be wrapped in a transaction, even reads (in many setups like Spring Data).

JPA vs JDBC
JPA: High (object-based)
JDBC: Low (manual SQL)

Insight:
JPA simplifies database access with object mappings and declarative queries, but understanding what happens under the hood (SQL generation, transaction boundaries, fetch types) is critical for real-world performance and reliability.
- -

#JPA ORM annotations

Core Concepts
1. @Entity
Marks a class as a JPA entity (must have a no-arg constructor).
Mapped to a database table.
Must be a top-level, non-final, non-abstract class.

2. @Table(name = "table_name")
Optional; used to specify a custom DB table name.
If not provided, table name defaults to class name.

3. @Id
Marks the primary key field of the entity.
Must be unique and non-null.

4. @GeneratedValue
Auto-generates ID values.

Common strategies:
AUTO – Let provider choose (default)
IDENTITY – Uses DB auto-increment
SEQUENCE – Uses a DB sequence
TABLE – Uses a table to simulate sequences

5. @Column

Maps a field to a specific DB column.

Can customize:
name
length
nullable
unique

6. @Transient

Excludes the field from persistence.
Not to be confused with Java's transient keyword.

7. @Temporal

Specifies the date/time precision for java.util.Date or Calendar.
DATE, TIME, TIMESTAMP

8. @Enumerated

Persists enums in DB.
EnumType.ORDINAL (default): saves index (unsafe!)
EnumType.STRING: saves enum name (safer)

9. @Lob

Used to store large data (CLOB/BLOB) like long text or images.

Relationship Annotations

10. @OneToOne

Maps one-to-one relationship.
Default fetch type: EAGER
Use mappedBy on non-owning side.

11. @OneToMany

One parent to many children.
Must use mappedBy and often needs @JoinColumn if bidirectional.
Default fetch type: LAZY

12. @ManyToOne

Many children to one parent.
Default fetch type: EAGER

13. @ManyToMany

Requires a join table.
Easily leads to performance issues if not managed well.
Default fetch type: LAZY

14. @JoinColumn

Defines the foreign key column in a relationship.
Controls the actual column name used for joins.

15. @JoinTable
Specifies a join table for @ManyToMany or custom join mapping.

Relationship, Default Fetch:
@ManyToOne : EAGER
@OneToOne : EAGER
@OneToMany : LAZY
@ManyToMany : LAZY

Missing mappedBy can cause extra join tables or insert/update errors.
Using EnumType.ORDINAL breaks when enum order changes.
@Lob fields can affect performance — avoid loading large blobs eagerly.
@Column(nullable = false) is schema-level enforcement, not runtime validation.

Summary:
JPA ORM annotations define how Java fields map to database schema. While they simplify persistence, they can easily introduce subtle bugs or performance issues if fetch types, cascading, or relationships are misconfigured.
- -

#Introduction to Spring Data

Spring Data:
Spring Data is a part of the Spring ecosystem focused on data access abstraction.
It reduces boilerplate code for CRUD, query execution, and data persistence by using repositories and annotations.

Key Goals of Spring Data
Simplify data access layer.
Provide generic Repository interfaces for all persistence stores.
Minimize boilerplate and JDBC/ORM code.
Offer custom query generation via method names.

Main Modules with use
Spring Data JPA : ORM using JPA/Hibernate
Spring Data MongoDB : NoSQL access for MongoDB
Spring Data JDBC : Lightweight JDBC abstraction
Spring Data Redis : Access Redis store
Spring Data Commons : Shared infrastructure for all modules

Core Interfaces
CrudRepository<T, ID> : Basic CRUD operations
JpaRepository<T, ID> : Adds JPA-specific features like pagination and sorting
PagingAndSortingRepository<T, ID> : Adds pagination/sorting features
Repository<T, ID> : Marker interface — no methods defined

Query Method Naming
Spring Data allows you to write queries like:
List<User> findByEmailAndStatus(String email, String status);

It auto-generates the SQL/JPQL based on method names.
Behind the scenes, it parses the method and creates the query automatically.

Important Points
Spring Data is NOT only for JPA

It’s a generic data access abstraction; JPA is just one implementation.

Interface Magic
You write only interfaces, no implementations — Spring creates proxies at runtime.

Avoid Logic in Repositories
Repositories are for data access only, not business logic — use the Service layer for that.

Use @EnableJpaRepositories
Required when package scanning is manual or different from default.
Not needed if using @SpringBootApplication in the root package.

Use @Repository Annotation
Although optional when extending Spring Data interfaces, it helps with:
Exception translation
IDE understanding
AOP behavior

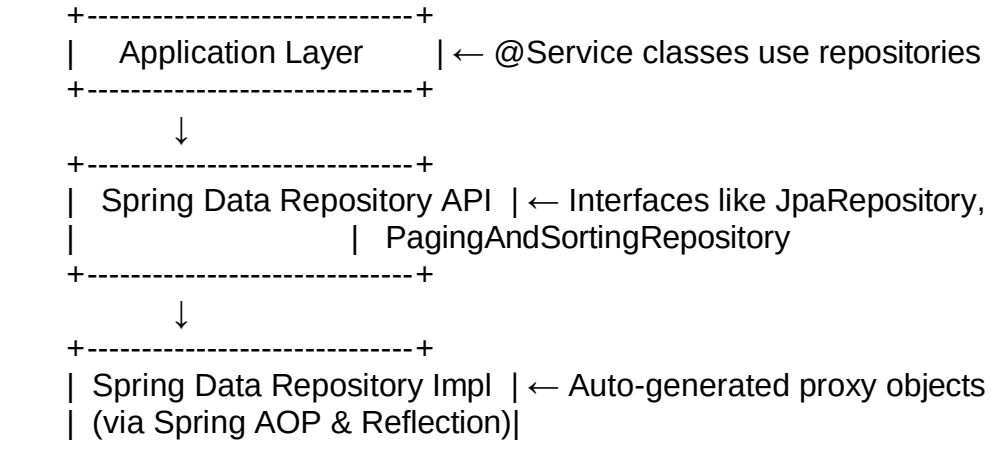
Real-World Insights
Spring Data JPA works best when your entities are well-designed and normalized.
For complex dynamic queries, prefer:
@Query annotation
Specifications
QueryDSL (for advanced cases)

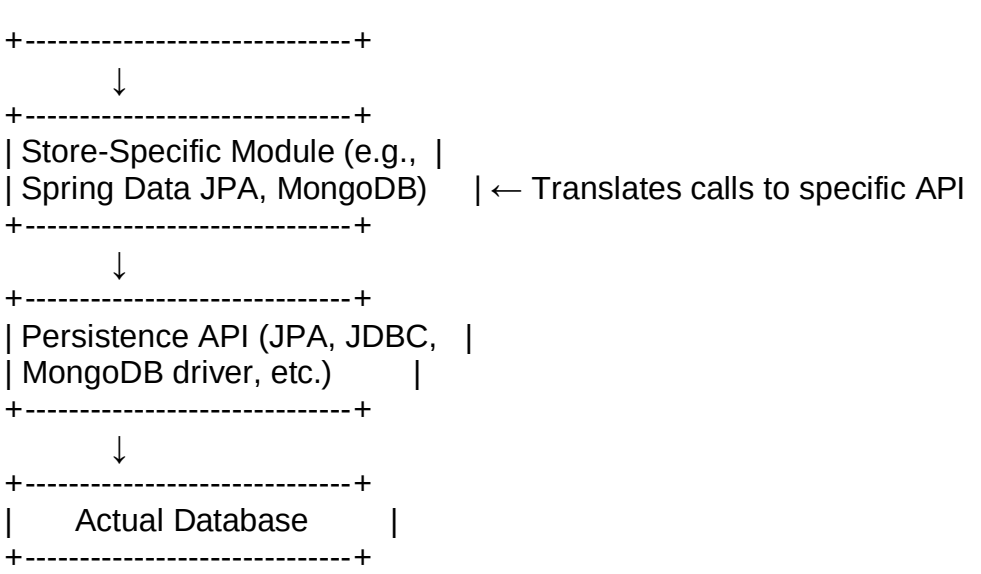
Summary:
Spring Data simplifies data access by generating repository implementations automatically. It’s powerful, but you must understand the method naming, query generation, and separation of concerns to use it effectively.
- -

#Spring Data architecture

Spring Data Architecture — Overview
Spring Data provides a layered architecture to abstract the data access logic from various data sources like JPA, MongoDB, Redis, Cassandra, JDBC, etc.

It uses:
Common infrastructure (Spring Data Commons)
Store-specific modules (e.g., JPA, MongoDB)
Repositories and runtime-generated implementations

Layered Architecture Breakdown




Key Components:
Spring Data Commons : Shared infrastructure (core interfaces, query parsing, etc.)
CrudRepository<T, ID> : Basic CRUD operations
PagingAndSortingRepository<T, ID> : Adds pagination & sorting support
JpaRepository<T, ID> : JPA-specific enhancements (e.g., batch operations)
Repository Proxy : Spring auto-generates implementation at runtime
@Query : Define custom JPQL/native queries
QueryDSL, Specifications : Advanced dynamic querying support

Tricky & High-Value Insights

No Class Implementation Needed

Just declare an interface — Spring injects a proxy as the implementation at runtime.

Query Methods Are Parsed at Startup

If you mess up a method like findByNameAndNonExistentField(), your app will fail to start.

Architecture Is Plug-and-Play

You can switch from JPA to MongoDB with minimal code changes — only repositories and configurations differ.

Query Resolution Order

Spring resolves queries in this order:
Declared @Query
Named query from entity
Query method name parsing

AOP + Reflection Magic

Spring uses AOP and CGLIB/Proxy to dynamically implement repository interfaces.
You never see the actual classes, but they’re fully functional at runtime.

Spring Boot Autoconfiguration

When using Spring Boot with spring-boot-starter-data-jpa, most of the architecture (like DataSource, EntityManagerFactory, TransactionManager) is auto-configured.

Layers:
Application : Uses Service/Controller layers
Repository Interface : Extends CrudRepository/JpaRepository
Repository Implementation : Injected via proxy
Store Module : Converts method to native operation
Persistence API : Real implementation details
Database : Data is persisted here

Insight:
Spring Data architecture is designed to be declarative, extensible, and store-agnostic, making it extremely powerful — but you must know the inner flow to debug and optimize effectively.
– –

#Spring Data Java SE application

What Is a Spring Data Java SE Application?
A Java Standard Edition (Java SE) application using Spring Data JPA
Runs in a non-web environment (e.g., console app)
Uses Spring's core container, JPA, and EntityManager to manage persistence

Key Components Needed:
ApplicationContext Loads Spring beans from config
@Entity : Defines JPA entities (POJOs mapped to tables)
@Repository : Spring Data interfaces for persistence
EntityManagerFactory : Low-level JPA factory (used under the hood)
DataSource : JDBC DB connection
JpaTransactionManager : Manages transactions via Spring

Typical Setup (No Spring Boot)
1. Persistence Configuration (XML or Java-based):
```
@Configuration
@EnableJpaRepositories(basePackages = "com.example.repo")
@EnableTransactionManagement
public class AppConfig {
   @Bean
   public DataSource dataSource() { ... }

   @Bean
   public LocalContainerEntityManagerFactoryBean emf() { ... }

   @Bean
   public PlatformTransactionManager txManager(EntityManagerFactory emf) {
      return new JpaTransactionManager(emf);
   }
}
```

2. Main Method (Runner):
```
public class MainApp {
   public static void main(String[] args) {
      AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);
      MyService service = context.getBean(MyService.class);
      service.doSomething();
      context.close();
   }
}
```

Notes:
1. Spring Data Works Without Web Server
Fully functional in Java SE, not just for web apps
Just need to load ApplicationContext manually

2. @Transactional Still Required

Transaction boundaries must still be declared, e.g., on service methods

3. No Autoconfiguration

Unlike Spring Boot, you must manually configure:
DataSource
EntityManager
TransactionManager

4. Repositories are Auto-Implemented

```
public interface ProductRepository extends JpaRepository<Product, Long> {
    Product findByName(String name);
}
```

Spring auto-creates implementation behind the scenes.

5. No @SpringBootApplication Shortcut

You must explicitly use:
@Configuration
@EnableJpaRepositories
@ComponentScan (if needed)

Mistake Issue:
Missing @EnableTransactionManagement : Transactions won't work
Forgetting to close ApplicationContext : Memory leaks in long-running apps
Using Spring Boot-style shortcuts : Not supported in plain Java SE

Example Console Output App:

```
@Service
public class MyService {
    @Autowired
    private UserRepository repo;

    @Transactional
    public void runLogic() {
        User user = new User("Amit", "amit@email.com");
        repo.save(user);
    }
}
```

Summary:
A Spring Data Java SE app is a clean, modular setup for managing JPA-based persistence without a web environment. Manual configuration replaces autoconfiguration, but Spring still powers dependency injection, transaction management, and repository abstraction.

#Transaction management using @Transactional

What is @Transactional?
@Transactional is a Spring annotation that manages declarative transaction boundaries, ensuring ACID (Atomicity, Consistency, Isolation, Durability) in your code.

@Transactional

```
public void transferMoney(...) {
```

```
    // multiple DB operations
}
```

Where Can You Use It?
~Typically on service layer methods
~Can be placed at:
Method level (preferred)
Class level (applies to all public methods)
Interface methods (but not recommended)

Default Behavior of @Transactional:
Propagation : REQUIRED (joins existing or creates new)
Isolation : DEFAULT (DB-specific)
Rollback : Only on unchecked exceptions
Read-only : false

Transactional Propagation Types:
Propagation : Behavior
REQUIRED : Joins current or starts new (default)
REQUIRES_NEW : Suspends existing and starts new transaction
NESTED : Savepoint within parent transaction (if supported)
SUPPORTS : Runs in transaction if exists
NOT_SUPPORTED : Always non-transactional
MANDATORY : Fails if no transaction exists
NEVER : Fails if transaction exists

Common Points:

1. Only Public Methods Get Proxied

Spring uses AOP proxies for @Transactional.
Internal calls like this.doSomething() don’t trigger it!

2. Rollback Occurs Only for RuntimeExceptions

```
@Transactional
public void someMethod() throws IOException {
    throw new IOException(); // No rollback by default!
}
```

Fix:
```
@Transactional(rollbackFor = IOException.class)
```

3. Calling Transactional Method from Same Class = Broken

```
// This will not apply @Transactional
this.someTransactionalMethod();
```

Solution: Move transactional logic to another bean or let Spring call it via proxy.

4. Read-Only Flag is Advisory

```
@Transactional(readOnly = true)
```

Optimizes performance for SELECTs
Some databases (like MySQL) ignore it
Won’t throw error if you write — just a performance hint

5. Checked Exceptions Don’t Trigger Rollback by Default

You must explicitly declare rollbackFor
RuntimeException → rollback
Exception → no rollback unless specified

Example:
```
@Transactional(propagation = Propagation.REQUIRED, rollbackFor = Exception.class)
public void updateOrderStatus(...) {
    // all DB ops here are atomic
}
```

Best Practices:
Use @Transactional on public service methods
Keep transactional methods short & focused
Use REQUIRES_NEW cautiously (commits even if outer tx fails)
Define rollback rules for checked exceptions
Use readOnly = true for read-heavy methods

Summary:
@Transactional enables powerful, declarative transaction control in Spring. Understand its proxy nature, rollback rules, and propagation types to avoid silent failures and maintain data integrity.

#Spring Repository (DAO) Layer

What is the Spring Repository (DAO) Layer?
The Repository Layer (aka DAO Layer):
Encapsulates persistence logic
Interfaces with the database using Spring Data JPA
Keeps the service layer free from SQL/JPQL logic

Core Annotations and Interfaces:
@Repository : Marks interface as a persistence component, enables exception translation
JpaRepository<T, ID> : Base interface that provides CRUD + pagination/sorting
CrudRepository<T, ID> : Simpler version of JpaRepository
PagingAndSortingRepository : Adds pagination & sorting to CRUD

Key Responsibilities:
CRUD Operations : Provided automatically (save(), findById(), delete(), etc.)
Custom Query Methods : Defined by method names (e.g., findByEmail())
JPQL/Native Queries : Defined using @Query annotation
Pagination/Sorting : Built-in with Pageable, Sort

Tricky and Critical Points

1. Don’t Add Logic Here

Repositories should contain only DB interaction
Move business rules to the Service Layer

2. Exception Translation

@Repository enables Spring to convert JPA exceptions into DataAccessException
If missing, you may not get meaningful exception wrapping

3. Spring Data Auto-Implementations

```
public interface UserRepository extends JpaRepository<User, Long> {
    User findByEmail(String email);
```

}

No implementation needed. Spring generates logic at runtime.

4. Use Optional<T> for Safe Return

Optional<User> findById(Long id);
Avoids null checks and makes null-handling explicit.

5. @Query Pitfalls
@Query uses JPQL by default (not native SQL).
Use nativeQuery = true for raw SQL.

@Query("SELECT u FROM User u WHERE u.status = ?1")
List<User> findByStatus(String status);

Common Mistakes:
Mistake Why It's a Problem
Mixing business logic into DAO Breaks separation of concerns
Using @Query when method names suffice Adds complexity unnecessarily
Not handling Optional properly Can cause NoSuchElementException
Using native SQL without nativeQuery=true Causes query parsing errors

Best Practices:
Keep repositories lean and declarative
Use Spring Data query methods when possible
Delegate complex logic to service layer
Use Pageable and Sort for scalable data retrieval
Define custom repository interfaces for advanced queries

Example:
```
@Repository
public interface ProductRepository extends JpaRepository<Product, Long> {
    List<Product> findByCategory(String category);

    @Query("SELECT p FROM Product p WHERE p.price > ?1")
    List<Product> findExpensiveProducts(double minPrice);
}
```

Summary
The Spring Repository layer is a declarative, type-safe way to handle database operations. It leverages interface inheritance, method name conventions, and annotations like @Query to eliminate boilerplate. Keep it focused, clean, and free of business logic.
- -

#Spring Service layer implementation

What is the Spring Service Layer?
The Service layer in Spring:
Holds business logic
Acts as a bridge between Controller (web layer) and Repository/DAO (data layer)
Handles transaction management, exception handling, and data orchestration

```
@Service
public class UserService {
```

```
    @Autowired
    private UserRepository userRepository;

    public User getUser(Long id) {
        return userRepository.findById(id).orElseThrow(...);
    }
}
```

Why Use a Service Layer?
Responsibility : Purpose
Business Logic : Implements rules/conditions beyond simple CRUD
Transaction Management : Controls atomicity and rollback using @Transactional
Reusability : Centralized logic for multiple controller/use cases
Security & Validation : Can apply @PreAuthorize, custom checks
Decoupling : Keeps controller and repository logic clean and separate

Typical Flow in Spring MVC:
Controller → Service Layer → Repository Layer → DB

Key Annotations:
Annotation : Description
@Service : Marks class as Spring-managed service bean
@Transactional : Enables transaction boundary (use at method/class level)
@Autowired : Injects repository or other service dependencies

Tricky Implementation Points

1. Services Should Be Stateless

Avoid keeping fields that store user/session-specific state.
Share logic across requests safely.

2. @Service Is Optional Technically, But Important

You could use @Component, but @Service provides semantic clarity and supports tool-based stereotypes (like AOP targeting).

3. Avoid Business Logic in Controllers

Bad:

```
userRepo.save(user); // directly in controller
```

Good:

```
userService.registerUser(user);
```

4. Use Interfaces (Optional but Recommended)

Enables easy testing, mocking, and future refactoring.

```
public interface UserService {
    User getUser(Long id);
}
```

5. Cross-Service Communication

Inject one service into another with care to avoid circular dependencies
Extract shared logic into utility classes or helper services

Best Practices:
Keep service methods focused and meaningful
Apply @Transactional where data integrity matters

Delegate CRUD to repository, not duplicate queries
Throw custom exceptions for business errors
Write unit tests using mock repositories

Example Service Implementation

```
@Service
public class ProductService {

    @Autowired
    private ProductRepository productRepo;

    @Transactional
    public Product updateStock(Long productId, int quantity) {
        Product product = productRepo.findById(productId)
                .orElseThrow(() -> new NotFoundException("Product not found"));
        product.setStock(product.getStock() + quantity);
        return productRepo.save(product);
    }
}
```

Summary:
The Spring Service Layer is the core of application logic. It isolates business rules, handles transactions, and mediates between web input and database operations. Implement it with @Service, use @Transactional wisely, and keep methods clean and testable.

#Spring Managed Service Layer to handle transactions.

What is the Spring-Managed Service Layer?
In a layered Spring application:
The Service layer holds business logic
It coordinates multiple DAOs/repositories
It's the best place to handle transactions

Why Use @Transactional in Service Layer?
Annotating service methods with @Transactional allows Spring to manage DB transactions automatically, ensuring data consistency and rollback on failure.

```
@Service
public class AccountService {

  @Transactional
  public void transferFunds(Long fromId, Long toId, double amount) {
    // logic to debit from one and credit to another
  }
}
```

Key Reasons to Handle Transactions in the Service Layer:
Business logic focus : Service layer combines multiple repository calls
Atomicity : All-or-nothing: If one step fails, rollback entire method
Separation of concerns : DAO = data access; Service = business + transaction
AOP compatibility : Spring uses proxies to wrap service methods

Transactional Propagation Levels:
Propagation : Behavior
REQUIRED : Default. Joins existing tx or creates new if none. most common
REQUIRES_NEW : Suspends existing tx, starts a new one
NESTED : Creates nested tx with savepoints (if DB supports it)
MANDATORY : Must be called inside a transaction, else exception
NOT_SUPPORTED : Runs outside any transaction
NEVER : Fails if there's an active transaction
SUPPORTS : Joins tx if exists, else runs non-transactionally

Points

1. Transactional Proxy Only Works on Public Methods

Spring AOP uses dynamic proxies, which only intercept:
Public methods
Called from outside the class

(this.myMethod() will not trigger @Transactional)

2. Transactional at Class vs Method Level

You can annotate the whole service class:
@Transactional
@Service
public class MyService { ... }
(Method-level annotations override class-level if both exist.)

3. Rollback Rules

By default, only unchecked (RuntimeException) causes rollback.
To rollback on checked exceptions:
@Transactional(rollbackFor = Exception.class)

4. Don't Put @Transactional on Repository Layer

Repositories should stay thin, focus on data access.
Service layer should control the transactional boundary.

5. Propagation.REQUIRES_NEW can cause isolation issues

It suspends the current transaction and starts a new one, which may lead to dirty reads or inconsistencies if not used carefully.

Best Practices
Use @Transactional in service layer only
Keep service methods short and cohesive
Prefer REQUIRED, avoid REQUIRES_NEW unless truly needed
Ensure public method scope for transaction support
Handle checked exceptions carefully with rollback rules

Summary:
The Spring-managed service layer is the correct place to define @Transactional boundaries, allowing Spring to automatically handle commit/rollback, ensure data consistency, and respect separation of concerns. Use propagation carefully and know proxy limitations.
- -

#JPA Id generators

What are JPA ID Generators?
JPA uses ID generators to automatically assign primary key values to entities when persisting them to the

database.

Annotation:
@Id
@GeneratedValue(strategy = ...)

Why ID Generation Matters:
Determines how IDs are assigned and ordered
Affects insert performance, batch operations, and DB portability
Poor choice can lead to exceptions or ID conflicts

Strategies of @GeneratedValue:
AUTO : Default; JPA provider chooses (based on DB dialect) (Unpredictable; not DB-portable)
IDENTITY : Uses DB auto-increment column. Simple, but breaks batch inserts
SEQUENCE : Uses DB sequence object (supported by Oracle, PostgreSQL). Best for scalability, supports batch inserts
TABLE : Simulates sequence using a table. Slowest, least preferred; portable but inefficient

@SequenceGenerator Example:
@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "my_seq")
@SequenceGenerator(name = "my_seq", sequenceName = "actual_seq_name", allocationSize = 50)

allocationSize: how many IDs to preallocate in memory
Higher = fewer DB hits, faster inserts
Lower = more DB hits, risk of gaps

@TableGenerator Example:
@Id
@GeneratedValue(strategy = GenerationType.TABLE, generator = "gen")
@TableGenerator(name = "gen", table = "id_gen", pkColumnName = "key", valueColumnName = "val", allocationSize = 1)

Stores last used ID in a table row
Useful if DB doesn’t support sequences or identity
Very slow, rarely used in modern apps

Points
1. IDENTITY Strategy Limitation
JPA can’t batch insert entities using IDENTITY (because ID is generated by DB after each insert).
Leads to 1 insert = 1 DB call.

2. SEQUENCE Strategy + allocationSize
Default is allocationSize = 50.
If you restart the app without updating the sequence, you may get ID collisions.

3. AUTO is not portable
On MySQL = IDENTITY
On PostgreSQL = SEQUENCE
On H2 = depends on dialect
(Use AUTO only in demos or test environments.)

4. Manual IDs? Use @Id only
If you want to set the ID yourself:
@Id

private Long id;
(Do NOT use @GeneratedValue.)

5. Hibernate-specific UUID support

```
@Id
@GeneratedValue
@GenericGenerator(name = "uuid", strategy = "uuid2")
private UUID id;
```

Useful for microservices or distributed systems
Not standard JPA

Common Mistakes:
Using AUTO in production (non-portable)
Forgetting to define @SequenceGenerator or @TableGenerator when using SEQUENCE/TABLE
Using IDENTITY with saveAll() and expecting batching
Not handling ID generation strategy when migrating between DBs

Summary:
JPA ID generators control how entities get their primary keys. Use SEQUENCE for efficiency, IDENTITY for simplicity, and avoid TABLE unless absolutely needed. Always set the right allocationSize and avoid AUTO in production.

#Spring data query methods

What are Spring Data Query Methods?
Spring Data allows you to define query logic by just naming methods in your repository interface. No SQL/JPQL or implementation needed — Spring auto-generates queries based on method names.

Basic Syntax:

```
List<User> findByName(String name);
```

Spring interprets findByName and builds a query like:

```
SELECT * FROM user WHERE name = ?
```

Common Query Method Prefixes
findBy : SELECT query
countBy : COUNT query
existsBy : Boolean check
deleteBy : DELETE operation
readBy / queryBy : Alternative to findBy

Examples:

```
User findByEmail(String email);
List<User> findByAgeGreaterThan(int age);
List<User> findByNameContaining(String keyword);
List<User> findTop3ByOrderBySalaryDesc();
```

Points

1. Method names are parsed at app startup

If you misname a field or use the wrong syntax, the app fails to start.

Example mistake: findByNmae() (misspelled field)

2. Case Sensitivity

Query keywords (e.g. And, Or, GreaterThan) are case-insensitive, but field names must match entity properties exactly (case-sensitive).

3. Limiting Results

```
User findFirstByOrderByIdDesc();     // 1 result
List<User> findTop5ByAgeGreaterThan(int age); // top 5
```

4. Sorting via Method Name

```
List<User> findByCityOrderByLastNameAsc();
```

5. Using Optional<T>

```
Optional<User> findByUsername(String username);
```

Avoids null, encourages safe null-checking.

6. Dynamic Queries → Use @Query or Specifications

If method names become too long or dynamic, switch to:

```
@Query("SELECT u FROM User u WHERE u.email = ?1")
```

Specifications/QueryDSL for complex logic

7. Projections

You can use DTO interfaces to return custom fields:

```
interface UserNameOnly {
  String getName();
}
List<UserNameOnly> findBy();
```

Query Execution Flow:

Spring parses the method name.
Builds JPQL based on your entity.
Executes the query at runtime using EntityManager.
Returns result (as list, optional, page, etc.)

Summary:

Spring Data Query Methods allow declarative, zero-SQL querying by parsing method names. They're fast to write, easy to maintain, but can become brittle if method names grow too long or field names change.

Introduction to Web Services:

A web service is a platform-independent, language-neutral, and standard-based method of communication between client and server applications on the web.
platform-independent: It works on any operating system (Windows, Linux, Mac, etc.). language-neutral: Written in any programming language (Java, Python, .NET, etc.).
standard-based: Uses common web standards like HTTP, XML, JSON, SOAP, WSDL, etc.

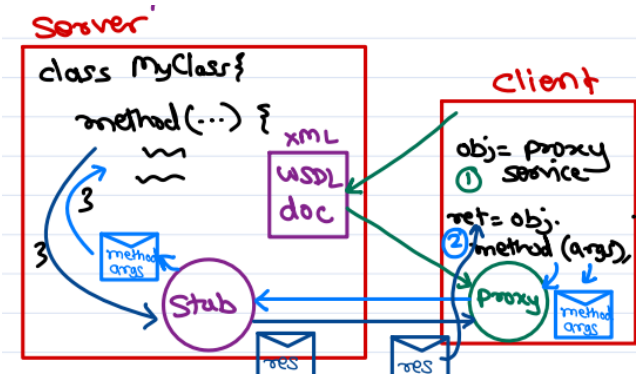
server: A **server** is an application or hardware that provides services to clients.(example: application server,web server, database server)
webserver: A web server is software that handles HTTP requests from clients (usually browsers), processes

them, and sends back the requested web pages or data.
Examples: Apache HTTP Server, Tomcat, Nginx.
web Application: A **web application** is a collection of web pages (HTML, CSS, JS) and backend logic running on a web server
static web pages:pages are fixed and always shows same content(eg.aboutus page)
dynamic webpaes: Pages that **change** based on user input, time, or data from a server.

Webservice:

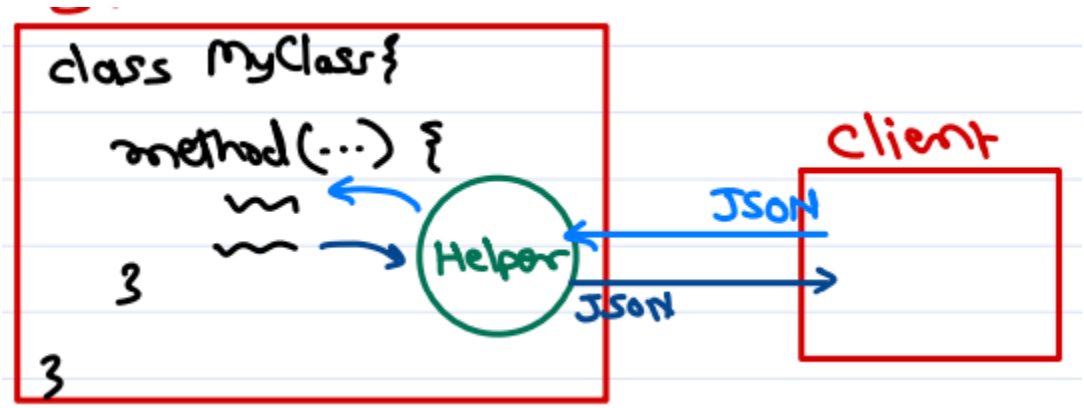
Two types of webservices

1) **SOAP (Simple Object Access Control**): Protocol-based web service using XML for message format.(Protocol based means It **defines exactly** how messages must be formatted (using XML), what headers to use, how to handle) it follows strict rules



- webserver like (apache ,tomcat,ngnix) hosts the webservice and stub is created at server side.(stub is helper code that listen for incoming request and calls actual function)
- WSDL(web service description language) file in xml also created on server side which define what webservice can do .
- WSDL document read by client and create proxy object at client side.client calls required method on proxy object,which internally send method and arg details to server side stub.
- Stub calls actual method and send result back to the client proxy .proxy return result to client.

2) **REST (Representation State transfer):** Architectural style using standard HTTP methods (GET, POST, etc.) and typically JSON. It provides **guidelines** or **principles** like statelessness, resource-based URIs, use of HTTP methods (GET, POST, etc



No validation of data like WSDL is done on client side so calling rest services at clent side is very lighter and faster.

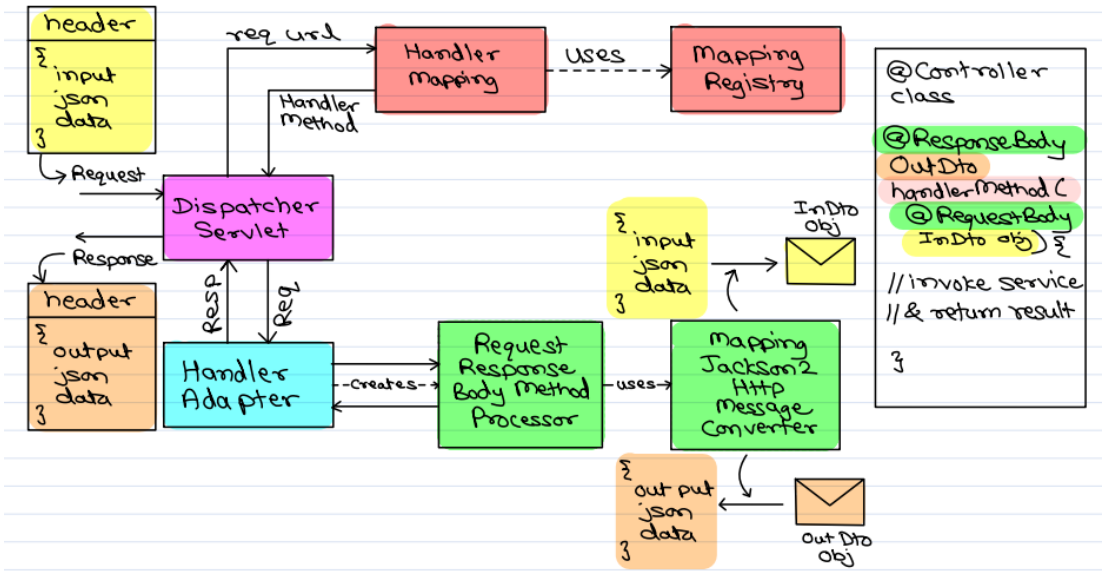
- REST is built on HTTP and HTTP follows request response model.
- HTTP protocol is stateless protocol (Each REST request is independent of another)
- REST is lightweight (than SOAP). No stub and proxy classes. No XML grammer (xsd) checks.
- Connectionless protocol connection is not kept connect permanently once response get connection get closed
- HTTP method in rest (GET,POST,PUT,PATCH,DELETE,HEAD,OPTIONS)
- REST API's client request (GET /students/101),
- server Response

```
{
 "id": 101,
 "name": "Madhuri",
 "marks": 92
}
```

- Response Status
  1) 1xx info
  2) 2xx success
  3) 3xxredirection
  4) 4xx client error
  5) 5xx server error

- **Content-Type** is an HTTP **header** that tells the server or client **what kind of data** is being sent or received in the **body** of the request or response.(text,image,audio,video,application/json)
- JSON is Java Script Object Notation. It is set of key-value pairs and supports few data types like numeric, string, boolean, null, array and objects.

REST ARCHITECTURE:



Every request first goes to **Dispatcher Servlet** (called the Front Controller).it decides where to send request.

- The request url is mapped to request handler method in controller by handler mapping bean .(handler mapping check which controller and method should handle the url) again like(/api/student maps to controller.getstudents(),and sent it to dispatcher servlet
- Handler adapter executes the selected method like getstudent()
- If the @Requestbody and @responsebody is used handle adapter creates RequestResponseBodyMethodProcessor bean . this bean internally use Jackson converter (MappingJackson2HttpMessageConverter bean) to convert request body json to required Java object (as given in method argument).
- Then request handler method is executed that may call service and dao layer It produces response Java object.
- Again RequestResponseBodyMethodProcessor bean internally use Jackson converter to convert Java object to Json format
- Finally this Json response is sent back to the client by DispatcherServlet.

Controller vs RestController

@Controller: is an annotation in Spring Framework.

- It marks the class as controller that handle web request
- Used in **Spring MVC** to build web applications with **views** (HTML pages, JSP, Thymeleaf, etc.).
- How controller works
  - When a user visits a URL, Spring looks for a controller method mapped to that URL.
  - The method runs, does some processing (like fetching data).
  - It adds data to a **Model** object (to pass data to the view).
  - It returns the **name of a view** (like "home").
  - Spring's **ViewResolver** looks for a template file with that name, like:

    JSP: /WEB-INF/views/home.jsp

    Thymeleaf: /templates/home.html
  - The template engine renders the page, inserting the data from the model.
  - The rendered HTML is sent back to the browser.

- To return raw data (like JSON), you need to add @ResponseBody on methods or use @RestController. It works together with other components like **ViewResolver**, **Model**, and **Templates**.
- For web apps with views. It return View (HTML pages) or raw data (with @ResponseBody)

@RestController: it is specialized version of @Controller in Spring.it combine @Controller and @ResponseBody automatically

- Used to build restful web services or APIs
- Every method returns **data (like JSON or XML)** directly in the HTTP response body
- No view rendering — just raw data responses.

For RESTFul APIs return Data directly (JSON/XML) in HTTP response body

@ReqeuestBody, @ResponseBody, ResponseEntity

@ReqeuestBody : It **binds the HTTP request body** (usually JSON or XML) to a Java object.

- Usually used in POST, PUT, PATCH requests where the client sends JSON or XML data.
- Spring uses **HttpMessageConverters** (like Jackson JSON converter) behind the scenes.to convert Jsonor Xml into Java
- If the incoming data does not match the Java object structure or is malformed, Spring throws exceptions like HttpMessageNotReadableException
-

@ResponseBody is a Spring MVC annotation that tells the framework: "Take the return value of this method and write it directly to the HTTP response body.

- Instead of rendering a view (like a JSP or Thymeleaf template), the data is serialized (usually to JSON or XML) and sent as the raw response content.
- When a controller method is annotated with @ResponseBody, Spring skips the view resolution step.
- Spring uses **HttpMessageConverters** to convert the returned Java object to the appropriate format based on the client's Accept header or default content type.
- Use @ResponseBody on controller methods when you want to return **raw data** (JSON, XML, plain text, etc.) instead of a view.

@ResponseEntity : **ResponseEntity<T>** is a class in Spring Framework used to represent the **entire HTTP response**, including:

- **Status code** (e.g., 200 OK, 404 Not Found)
- **Headers** (e.g., Content-Type, Authorization)
- **Body** (e.g., JSON, text, file)
- It gives **full control** over the HTTP response returned from a controller method.

- Building Restful Web Services with Spring Boot

1. Create new Spring Starter project with dependencies Dev Tools, Lombok, JDBC API, MySQL Driver, and Spring Web.
2. In application.Properties add database settings.
3. Create @RestController -Rest api (Handling HTTP request from client ,Mapping URLs to methods,sending responses in JSON,XML,communicating with service layer)

4. Create Entity class (eg.User.java,POJO Class)
5. Create repository (UserDao interface,UserDao impl class) which extends by JPA repository(interacting with database)
6. Create Service class(business logic and validation)

```
Client (e.g. Postman/React frontend)
   ↓
Controller → (handles HTTP requests)
   ↓
Service → (business logic, validation)
   ↓
DAO / Repository → (interacts with DB)
   ↓
Database
```

Exception handling:

Ways to Handle Exceptions in Spring Boot:

1. Using @ControllerAdvice + @ExceptionHandler

This is the recommended approach for global exception handling.

2. Using @ExceptionHandler in Controller class

For local exception handling (less common).

3. Using ResponseStatusException

For throwing built-in HTTP exceptions.

1.Global Exception Handling:

Step 1. Create Custom Exception class

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

Step2.Create Global Exception handler:

```java
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(UserNotFoundException.class)
    public ResponseEntity<String> handleUserNotFound(UserNotFoundException ex) {
        return new ResponseEntity<>(ex.getMessage(), HttpStatus.NOT_FOUND);
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<String> handleAllExceptions(Exception ex) {
        return new ResponseEntity<>("Internal Server Error: " + ex.getMessage(),
                                    HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

Step 3:Throw exception in service

```java
public User getUserById(Long id) {
    return userRepository.findById(id)
        .orElseThrow(() -> new UserNotFoundException("User ID " + id + " not found"));
}
```

2. ResponseStatusException

```java
@GetMapping("/{id}")
public User getUser(@PathVariable Long id) {
    return userRepository.findById(id)
            .orElseThrow(() -> new ResponseStatusException(
                HttpStatus.NOT_FOUND, "User not found"));
}
```

Generating standard/consistent responses:

To make your API responses clean and consistent (success & error), it's best to **wrap all responses in a standard format** — whether it’s success or failure.

```java
@Data
@RequiredArgsConstructor
@AllArgsConstructor
public class ResponseUtil <T> {

        private String status;
        private String message;
        private T data;


        public static <T>ResponseUtil<?> apiSuccess(T data){

                ResponseUtil<T> result = new ResponseUtil<T>("success",null,data);
                return result;
        }

        public static <T>ResponseUtil<?> apiError(String message){
                ResponseUtil<T> result = new ResponseUtil<T>("error",message,null);
                return result;
        }



}
```

A **standard response format** means that every response your REST API sends back to the client — whether success or error — follows a **consistent structure**. This typically includes fields like:

- Status or success indicator (success or status)
- Actual data (data)
- Error details (if any) (error or message)
- Timestamp or metadata (timestamp)

This consistency allows clients to parse and understand responses easily, without having to write different code for different endpoints or error cases

Global Exception Handling

- Use @RestControllerAdvice for centralized exception handling.
- Catch specific exceptions (e.g., UserNotFoundException) and return a formatted error response.
- Catch general exceptions as fallback with a generic error message.
- This avoids repetitive try-catch blocks and provides consistent error output.

```java
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(UserNotFoundException.class)
    public ResponseEntity<ApiResponse<Object>> handleUserNotFound(UserNotFoundException ex) {
        ApiResponse<Object> errorResponse = new ApiResponse<>("error", null, ex.getMessage());
        return new ResponseEntity<>(errorResponse, HttpStatus.NOT_FOUND);
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<ApiResponse<Object>> handleOtherExceptions(Exception ex) {
        ApiResponse<Object> errorResponse = new ApiResponse<>("error", null, "Internal Server Error"
        return new ResponseEntity<>(errorResponse, HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

Session 13 & 14:

- Detailed flow of Spring REST service execution

1.Client Sends HTTP Request:

- A client (browser, mobile app, or another service) sends an HTTP request to the Spring Boot REST API.
- The request includes URL, HTTP method (GET, POST, etc.), headers, and possibly a body (for POST, PUT).

1. **Request Reaches DispatcherServlet (Front Controller)**

- Spring Boot uses the **DispatcherServlet** as the **front controller**.
- DispatcherServlet is a Servlet that intercepts all incoming HTTP requests (configured automatically in Spring Boot).
- It acts as the central entry point to the Spring MVC framework

2. Handler Mapping: Find Appropriate Controller
    - DispatcherServlet asks one or more **HandlerMapping** beans to find a matching **Handler** (usually a Controller method) based on:
    - Request URL path
    - HTTP method (GET, POST, etc.)
    - Request parameters and headers (if applicable)
    - If no match is found, Spring returns 404 Not Found.

3. Handler Adapter: Invokes Controller Method

- Once the HandlerMapping locates the controller method, DispatcherServlet uses a **HandlerAdapter** to invoke that method.
- HandlerAdapter abstracts how to call the handler method, including preparing method parameters.
-

4. Controller Method Execution
    - The selected controller method executes business logic, typically:
    - Calling service layer methods
    - Retrieving data from the database via repositories.
    - Performing validations.
    - Constructing a response object.

    - Method parameters are automatically resolved by Spring from:
    - Path variables (@PathVariable), Query parameters (@RequestParam), Request body (@RequestBody), Headers (@RequestHeader)
5. Request Body and Response Body Processing

- If the controller method parameter is annotated with @RequestBody, Spring uses **HttpMessageConverters** to deserialize the incoming JSON/XML body into a Java object.
- If the method returns an object and is annotated with @ResponseBody or the controller is @RestController, Spring uses HttpMessageConverters to serialize the Java object back to JSON/XML for the HTTP response.

6. Validation and Exception Handling
    - If validation annotations (e.g., @Valid) are used, Spring automatically validates input and throws exceptions if invalid.
    - Exceptions during processing are caught by:
    - **@ExceptionHandler** methods inside controllers.
    - Or globally by @ControllerAdvice annotated classes.
    - Exception handlers produce a proper error response (standardized if configured)

7. Response Returned
    - After controller execution and response serialization, DispatcherServlet sends the HTTP response back to the client.
    - The response includes:
    - HTTP status code (200, 400, 404, etc.)
    - Response headers (Content-Type, etc.)
    - Response body (JSON/XML or other media type)

```
Client Request
     ↓
DispatcherServlet (Front Controller)
     ↓
HandlerMapping (Find controller method)
     ↓
HandlerAdapter (Invoke controller method)
     ↓
Controller Method
     ↓
(Service layer, Repository called)
     ↓
Return Java object (response)
     ↓
HttpMessageConverter (Serialize to JSON/XML)
     ↓
DispatcherServlet sends HTTP Response
     ↓
Client receives response
```

Content Negotiation: is a mechanism used in RESTful web services to determine the **media type (format)** of the response that should be returned to the client.
It simple means choosing the **format** of the data (like JSON or XML) that the server sends in response to a client request.
Different clients may want data in different formats:

- A mobile app may want **JSON**
- A legacy system may want **XML**

Spring Boot supports **content negotiation** using:

- Request headers (Accept)
- URL extensions (.json, .xml) — optional
- Request parameters — optional

Request:
GET /api/user/1
Accept: application/json

Response (JSON):
{
  "id": 1,
  "name": "John"
}

Versioning RESTful Services:

- API Versioning means supporting multiple versions of your REST API to avoid breaking existing clients when your backend changes

Why Versioning

- Your mobile app uses /api/users . Later, you change the structure of the response
- **Without versioning:** Older apps break
- **With versioning:** You keep /v1/api/users and add a new /v2/api/users

Ways to Version REST APIs in Spring Boot

1. **URI Versioning** :

- Version number is included in the URL

```
@RestController
@RequestMapping("/api/v1/users")  // Version in the URL
public class UserV1Controller {
    @GetMapping
    public String getUsers() {
        return "Version 1 User List";
    }
}
```

Here in URL in request mapping we can change the version (“api/v2/users)

1. **Request Parameter Versioning**

- **Version is passed as a query parameter**

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

    @GetMapping(params = "version=1")
    public String getV1() {
        return "User List V1";
    }

    @GetMapping(params = "version=2")
    public String getV2() {
        return "User List V2";
    }
}
```

2. Header Versioning:

- Version information is sent in custom HTTP headers

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

    @GetMapping(headers = "X-API-VERSION=1")
    public String getV1() {
        return "User List V1";
    }

    @GetMapping(headers = "X-API-VERSION=2")
    public String getV2() {
        return "User List V2";
    }
}
```

4.Content Negotiation Versioning

- Uses the Accept header with a custom media type

```java
@RestController
@RequestMapping("/api/users")
public class UserMediaTypeVersionController {

    @GetMapping(produces = "application/vnd.company.app-v1+json")
    public String getV1() {
        return "User List V1";
    }

    @GetMapping(produces = "application/vnd.company.app-v2+json")
    public String getV2() {
        return "User List V2";
    }
}
```

Why is Versioning Important?

- To maintain **backward compatibility**
- To allow **continuous evolution** of the API
- To avoid breaking changes for existing users
- To support multiple clients (old app, new app, third-party)

Unit testing of Repository layer:

- RepositoryLayer is data access layer that interacts with the database.
- Typically uses Spring Data JPA repositories (JpaRepository, CrudRepository, etc.).
- Responsible for CRUD operations on entities.

Why Unit Test the Repository Layer?

- To ensure that database queries work correctly.
- To verify custom queries (JPQL, native SQL) behave as expected.
- To catch errors early in data access logic.

How to Unit Test Repositories?

- Use Spring Boot Test support with @DataJpaTest annotation.
- It sets up an in-memory database (like H2) automatically.
- Tests run fast and are isolated from the main database.

Unit testing of Controller Layer Mocking Service layer
What is Controller Layer Unit Testing?

- The Controller layer handles HTTP requests and returns responses.
- Unit testing controllers means testing controller logic in isolation from the service and repository layers.
- This ensures the controller handles requests, responses, validation, and error handling correctly.

Why Mock the Service Layer?

- The service layer contains business logic and dependencies (like repositories).
- For unit tests of controllers, mocking the service layer:
    - Isolates controller behavior.
    - Avoids dependency on business logic or databases.
    - Makes tests faster and more focused.

Typical Workflow

- Annotate test class with @WebMvcTest(MyController.class)
- Mock service layer with @MockBean
- Inject MockMvc for request simulation
- Use MockMvc to send HTTP requests and assert responses
- Set up expected mock responses via Mockito when()

Integration testing:

What is Integration Testing?

- Integration testing verifies the interaction between multiple components or layers of an application.
- Unlike unit testing (which tests components in isolation), integration tests check if modules work correctly together.
- For a Spring Boot REST app, this typically means testing the controller, service, and repository layers together, often with a real or in-memory database.

Purpose of Integration Testing

- To ensure that the application components are properly wired.
- To verify that HTTP endpoints, business logic, and persistence work as expectedtogether.
- To detect issues in configuration, data access, serialization, and other cross-cutting concerns.
- To simulate real use cases more closely than unit tests.

How to Write Integration Tests in Spring Boot?

- Use the @SpringBootTest annotation to load the full Spring context.
- Optionally configure a test database (H2 in-memory is common).
- Use **TestRestTemplate** or **MockMvc** to simulate HTTP requests.
- Can verify the full request-to-response flow including DB state changes.