SPRINGBOOT

**Auto-Configuration:**

**Auto-configuration** is the mechanism by which Spring Boot attempts to automatically configure your Spring application **based on the libraries present in the classpath**. It tries to provide sensible defaults for most use cases.

**How It Works**

1. **Classpath Scanning**: Spring Boot checks what dependencies are present.
   * Example: If spring-boot-starter-web is present, Spring Boot assumes you want a web application and configures DispatcherServlet, Tomcat, and other necessary beans automatically.
2. **Application Properties**: Auto-configuration can be customized through properties in application.properties or application.yml.
   * Example: spring.datasource.url, spring.datasource.username, etc., for database configuration.

**Normal Class , POJO and Java Beans**

**1. Normal Java Class**

* Any class you write in Java is a **normal class**.
* No specific conventions or rules are enforced.
* Can have any fields, methods, access modifiers, or constructors.

**Example:**

public class Car {

int speed;

String color;

public void drive() {

System.out.println("Car is driving");

}

}

* No restrictions.
* Not necessarily used to represent data.

**2. POJO (Plain Old Java Object)**

* A **POJO** is a simple Java object **without any special restrictions**, interfaces, or frameworks.
* Typically used to **store data**.
* No requirement for getter/setter methods (but often used).
* Doesn’t implement any specific interface like Serializable unless needed.

**Example:**

public class Car {

private String color;

private int speed;

// Constructor

public Car(String color, int speed) {

this.color = color;

this.speed = speed;

}

// Getters

public String getColor() { return color; }

public int getSpeed() { return speed; }

}

* **POJO = simple data object**, framework independent.

**3. JavaBean**

* A **JavaBean** is a **special type of POJO** with conventions:
  1. **Private fields** (encapsulation)
  2. **Public getter and setter methods**
  3. **No-argument constructor**
  4. Should be **serializable** (optional but recommended)

**Example:**

import java.io.Serializable;

public class Car implements Serializable {

private String color;

private int speed;

// No-argument constructor

public Car() {}

// Getter and Setter

public String getColor() { return color; }

public void setColor(String color) { this.color = color; }

public int getSpeed() { return speed; }

public void setSpeed(int speed) { this.speed = speed; }

}

**Key Differences**

| **Feature** | **Normal Java Class** | **POJO** | **JavaBean** |
| --- | --- | --- | --- |
| Fields | Any access | Usually private | Private |
| Methods | Any | Any | Public getters/setters |
| Constructor | Any | Any | Must have no-arg constructor |
| Serializable | Optional | Optional | Recommended (implements Serializable) |
| Purpose | General | Simple data holder | Framework-compatible data holder |

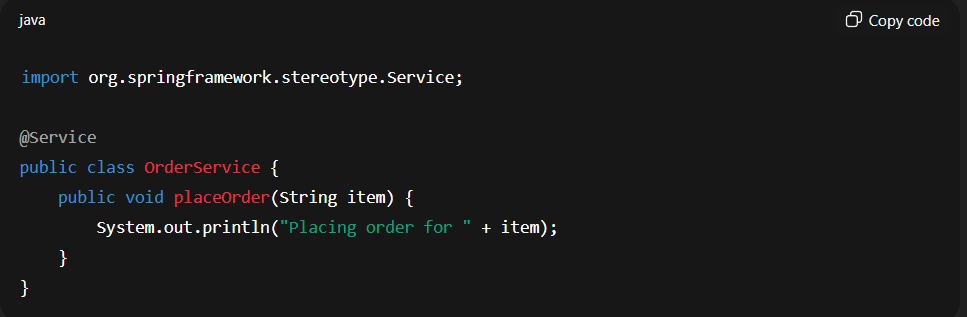
**✅ Summary**

* **Bean** = Object managed by Spring container.
* **Ways to create beans in Spring Boot**:
  1. @Component / @Service / @Repository / @Controller
  2. @Bean method in @Configuration class
* **Injection** = @Autowired (constructor, field, or setter)
* Spring manages the **lifecycle, scope, and dependencies** automatically.

**@component vs @Service**

**✅ Rule of Thumb**

* If the class is **business logic**, use @Service.
* If the class is **general purpose or helper**, use @Component.
* Functionally, Spring treats them **the same**; the difference is mainly **semantic clarity**.



**application.properties**

A **flat key-value configuration file** used by Spring Boot to define application settings such as server port, database connection, logging level, and other properties.

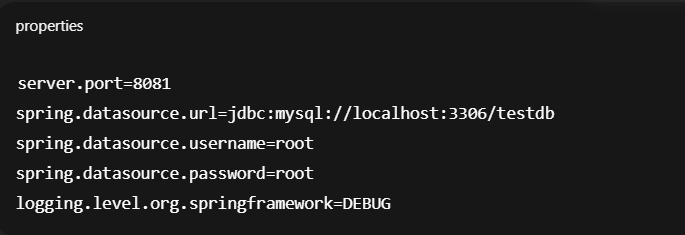
 **Format:** Key-value pairs.

 **Purpose:** Store configuration settings for Spring Boot apps.

 **Default location:** src/main/resources/.

 **Use cases:** Database connection, server port, logging level, etc.

Note: Each is written in the next line.



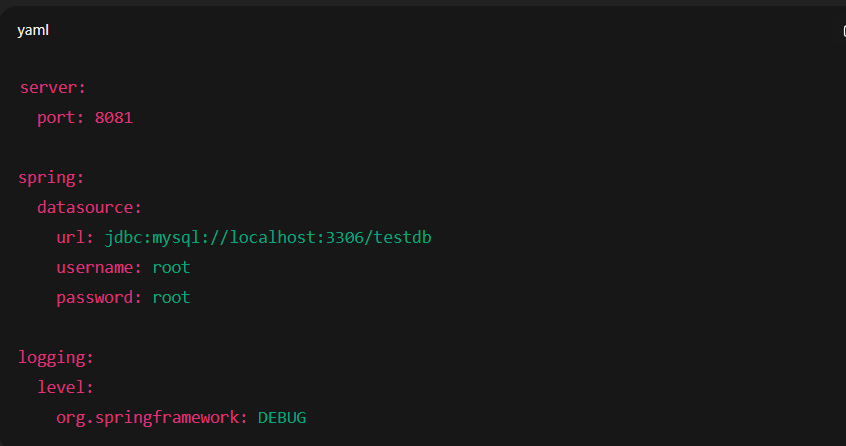
**2. application.yml**

A **YAML-based configuration file** used by Spring Boot for the same purpose as application.properties, but with a **hierarchical structure**, making it easier to manage complex or nested configurations.

 **Format:** YAML (hierarchical structure, indentation-based).

 **Purpose:** Same as application.properties, but better for **nested or complex configuration**.

 **Default location:** src/main/resources/.



**3. @SpringBootApplication / Application Class**

The **main entry point of a Spring Boot application**, annotated with @SpringBootApplication, which enables **auto-configuration, component scanning, and configuration**, and is responsible for **bootstrapping the Spring context**.

**IOC**

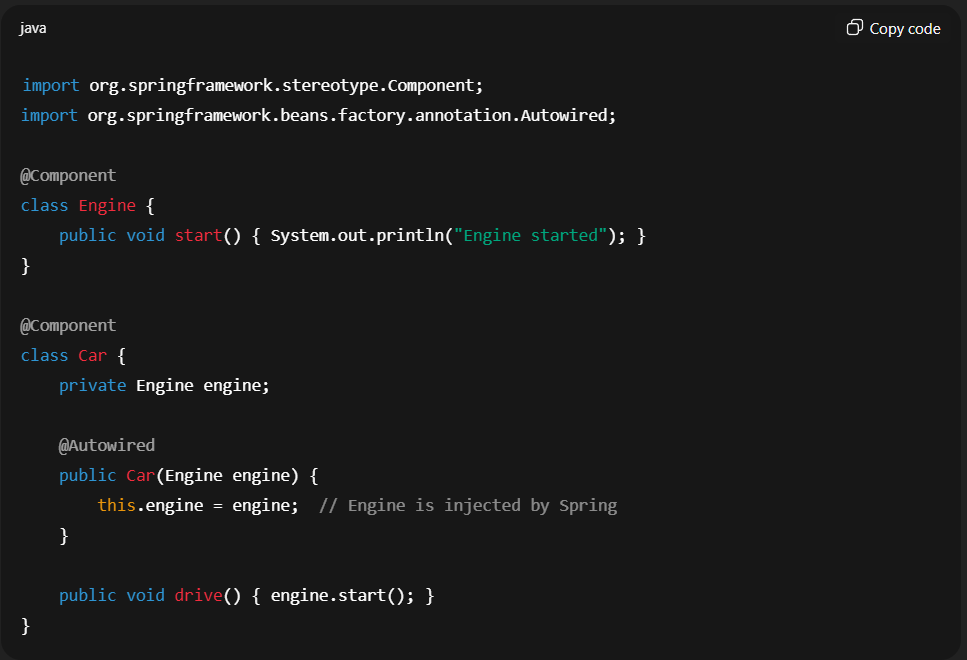
**Inversion of Control (IoC)** is a design principle in which the **control of object creation and dependency management is transferred from the program itself to a framework or container**.

In simpler words:

* Instead of the class creating its dependencies, **the framework (Spring) injects the required objects**.
* This **decouples the components**, making the code more modular, testable, and maintainable.

**How IoC Works in Spring**

1. You define **beans** (classes) in the Spring container.
2. You declare **dependencies** in your classes (via @Autowired, constructor, or setter).
3. **Spring container injects the required beans** at runtime.



 Here, Car **does not create an Engine instance itself**.

 The **Spring container** provides it → **IoC in action**.

**Benefits of IoC**

* Reduces **tight coupling** between components
* Encourages **modular and reusable code**

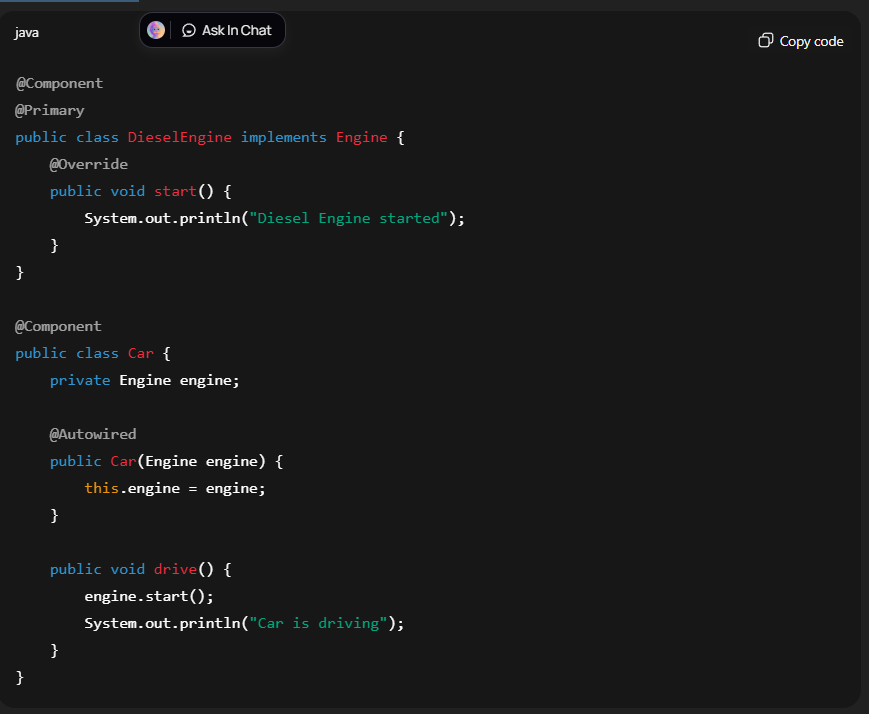
**Tight Coupling :**

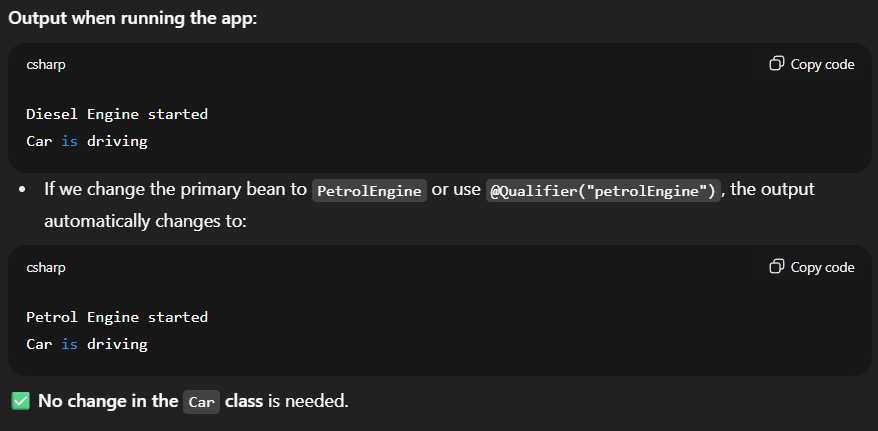
Tight coupling occurs when a class depends directly on the concrete implementation of another class, making the code less flexible and harder to maintain or test.

**IOC vs Tight coupling:**

**1. Output using Spring IoC (loose coupling)**

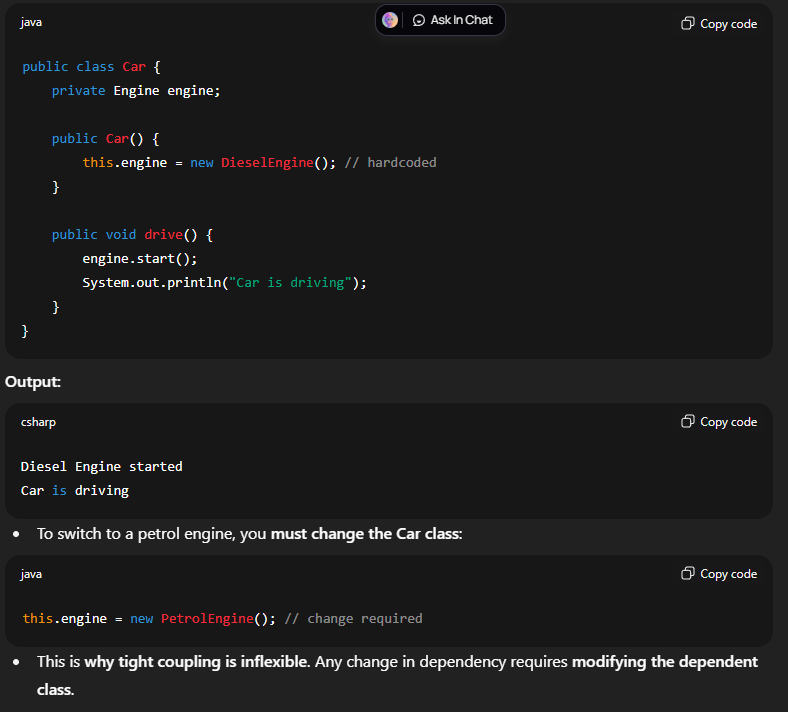
Suppose we have:





**2. How this would work with tight coupling?**

Tight coupling means Car **creates the engine itself**:



NOTE: In tight coupling the object is created in the class

Car c = new PetrolEngine();

But in IOC it can be using @Primary or @Qualifier

**🔹 Dependency Injection (DI)**

**Definition:**  
Dependency Injection (DI) is a **design pattern** used in Spring (and other frameworks) where the **Spring IoC container supplies the objects (dependencies) that a class needs**, instead of the class creating them itself.

👉 In simple terms:

* **Without DI (manual creation):** A class creates its own dependencies using new.
* **With DI (Spring way):** Spring creates and provides the required dependencies automatically.

**Relationship Between IoC & DI**

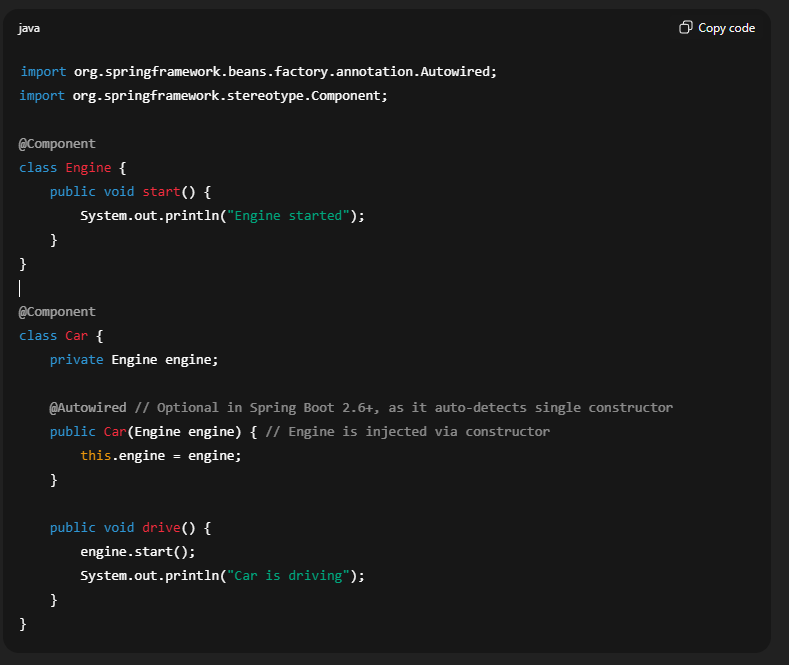
* IoC is the **principle** (big picture).
* DI is the **technique** (how IoC is achieved).

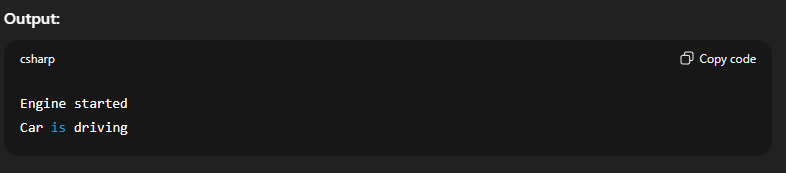
**Types of DI in Spring**

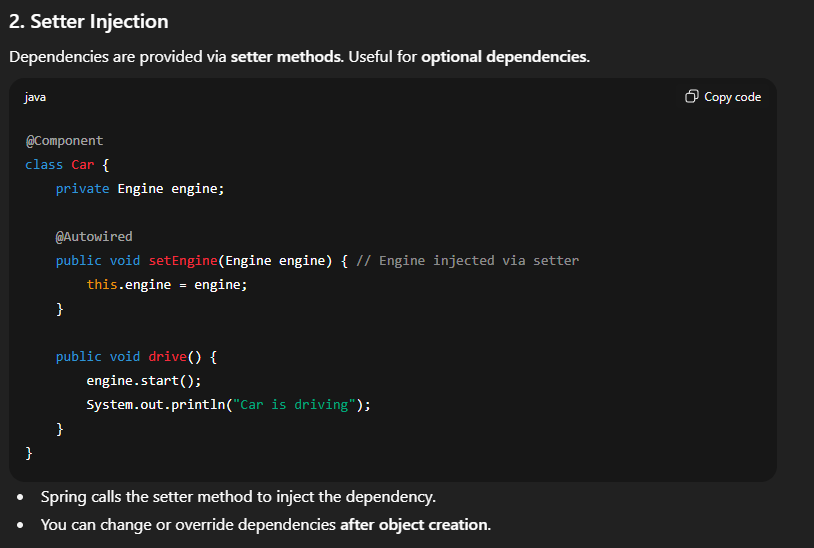
1. **Constructor Injection** → Dependencies passed via constructor
2. **Setter Injection** → Dependencies passed via setter methods
3. **Field Injection** → Dependencies injected directly into fields (using @Autowired)

1. Constructor Injection:

Dependencies are provided via the **constructor**. Recommended for **mandatory dependencies**.









**Comparison of Injection Types**

| **Type** | **When to Use** | **Pros** | **Cons** |
| --- | --- | --- | --- |
| Constructor Injection | Mandatory dependencies | Immutable, testable | Slightly more code |
| Setter Injection | Optional dependencies | Can change dependency later | Mutable, less strict |
| Field Injection | Quick and simple | Minimal code | Harder to test, uses reflection |

**Autowired**

@Autowired is an annotation in Spring/Spring Boot used for **automatic dependency injection**.  
It tells Spring’s **IoC container** to **inject a bean** (object) into another bean automatically, without you creating it manually using new.

**How it Works**

* Spring scans classes annotated with @Component, @Service, @Repository, or @Controller (or beans defined via @Bean in config).
* These classes become **beans** managed by Spring.
* When you put @Autowired on a field, constructor, or setter → Spring looks for a **matching bean by type** and injects it.

NOTE:- It tells the spring to intitailze those object rather than initializing using the new keyword.

|

|

|

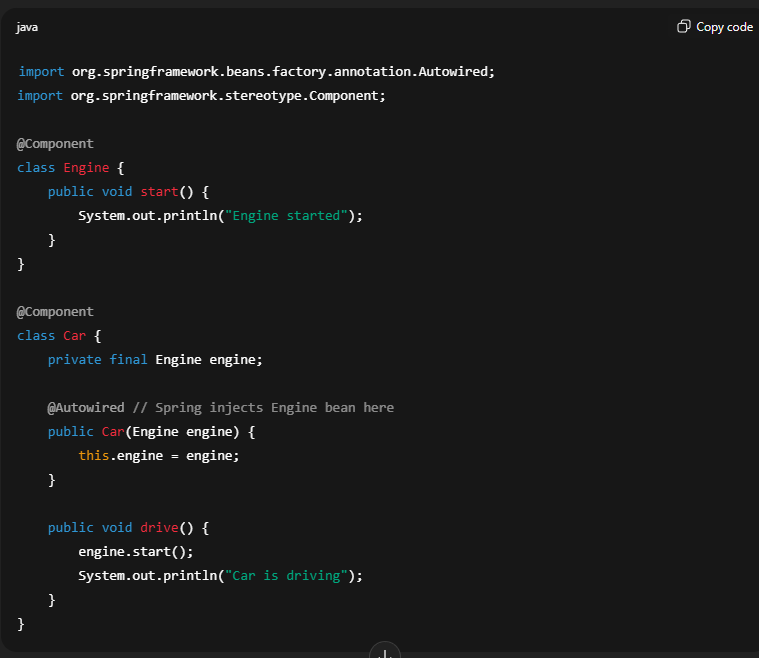
|

|

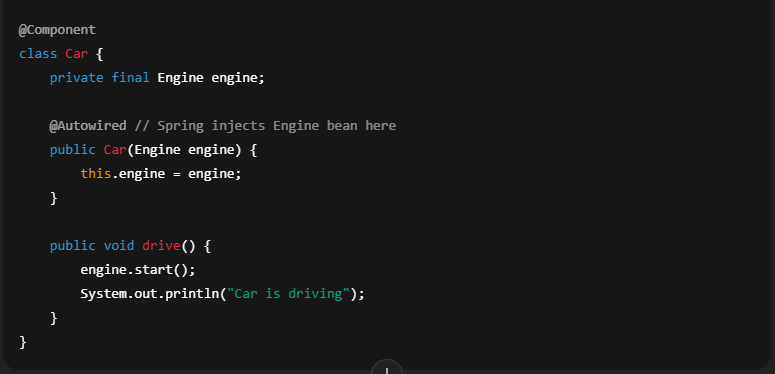
|

\ /

\/



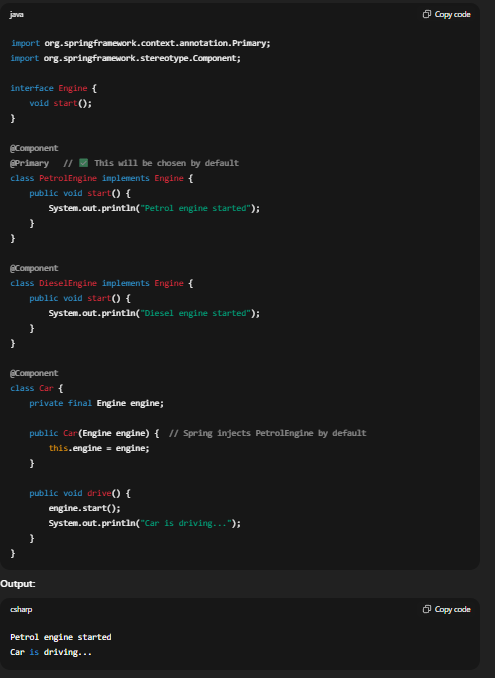
If more than one constructor exists then the spring will choose @Autowired constructor.



**@Primary v/s @Qualifier**

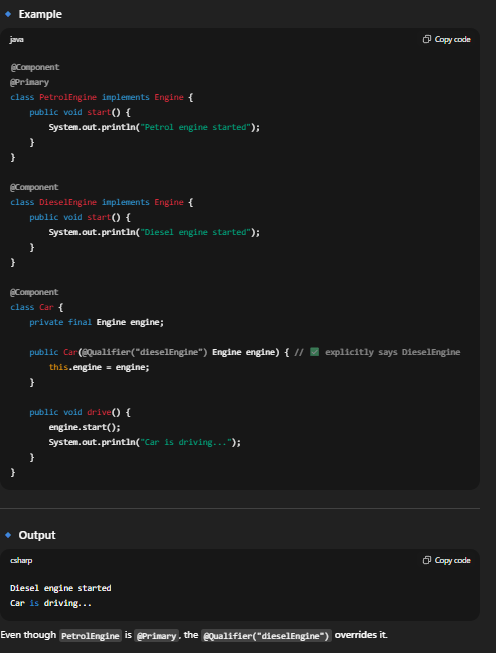
**🔹 @Primary**

* Tells Spring: **“If there are multiple beans of the same type, use this one by default.”**
* It’s like setting a **default bean**.



**🔹 @Qualifier**

* Used when you want to **explicitly specify which bean** to use, instead of relying on @Primary.
* Works like giving a bean a **name tag**.



Even though PetrolEngine is @Primary, the @Qualifier("dieselEngine") **overrides** it.

✅ **Rule of thumb:**

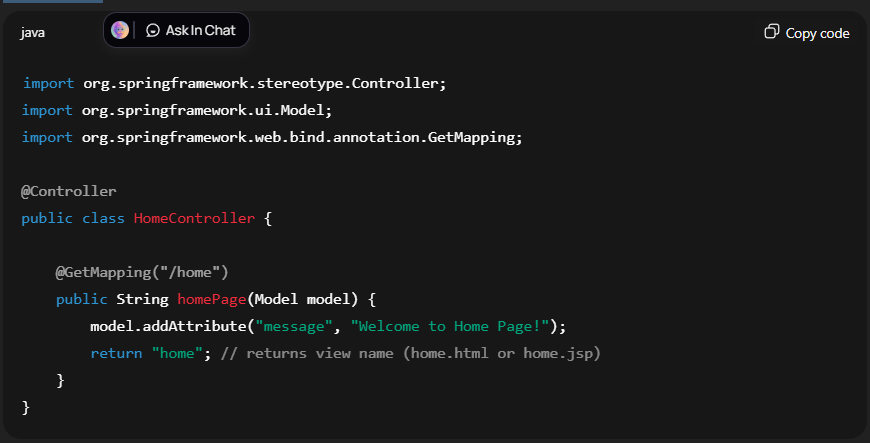
* If **only @Primary** → Spring picks that bean by default.
* If **@Qualifier is given** → it overrides @Primary.
* If **neither is given and multiple beans exist** → Spring throws NoUniqueBeanDefinitionException.

**Controller vs RestController**

**When to Use?**

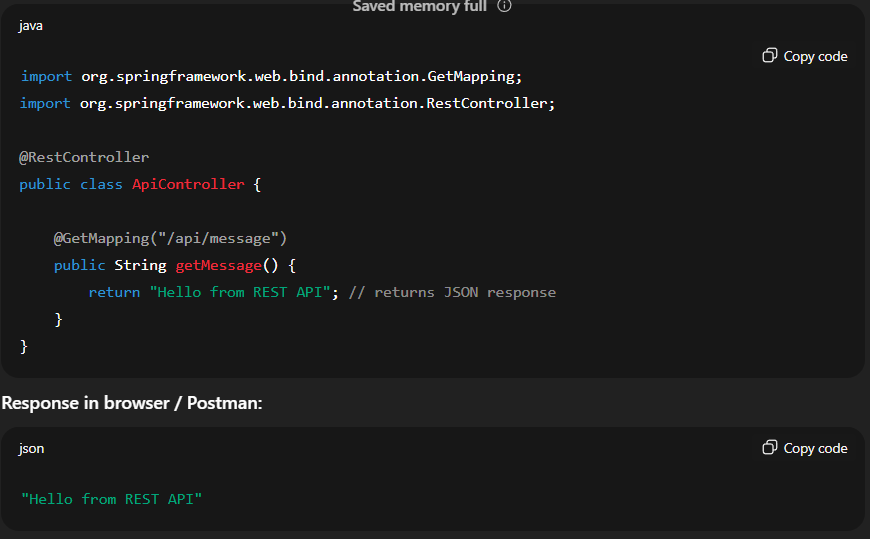
* Use **@Controller** if you’re building a **server-side rendered web app** (with Thymeleaf, JSP, etc.).
* Use **@RestController** if you’re building a **REST API** for frontend (React, Angular, mobile apps, Postman testing, etc.).

**1. @Controller**

* Used in **Spring MVC (Model-View-Controller)** applications.
* Returns **views (HTML, JSP, Thymeleaf, etc.)** instead of raw data.
* Typically used when building **web applications with UI**.
* Here, Spring looks for a template named home.html in templates/.

**2. @RestController**

* Shortcut for @Controller + @ResponseBody.
* Returns **JSON or XML** directly as a response (instead of views).
* Typically used when building **REST APIs**.



**🔹 Key Differences**

| **Feature** | **@Controller** | **@RestController** |
| --- | --- | --- |
| Purpose | For MVC web apps (returning views) | For REST APIs (returning data) |
| Returns | Views (HTML, JSP, Thymeleaf) | JSON/XML (via @ResponseBody) |
| Annotation shortcut | Just @Controller | @Controller + @ResponseBody |
| Usage | Web pages with UI | RESTful services (APIs) |

✅ **In short:**

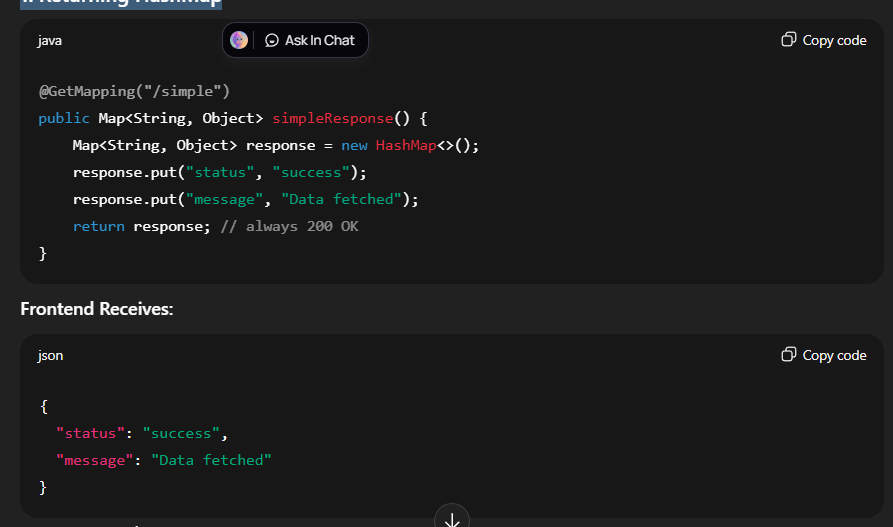
* @Controller → returns **views** (HTML).

@RestController → returns **data** (JSON/XML).

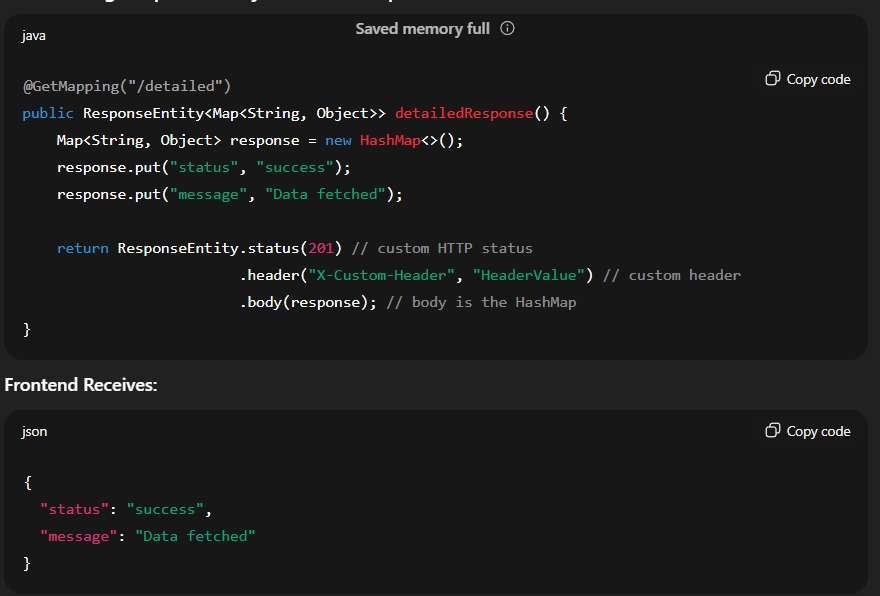
**HashMap vs ResponseEntity**

| **Feature** | **HashMap (or Map)** | **ResponseEntity** |
| --- | --- | --- |
| **Purpose** | Represents **data only** (body) | Represents **entire HTTP response** (body + headers + status) |
| **Status Code Control** | Always 200 OK (by default) | Can set **any HTTP status**: 200, 201, 400, 404, 500, etc. |
| **Headers** | Cannot set custom HTTP headers | Can set **custom headers** easily |
| **Serialization** | Converted to JSON automatically if returned from @RestController | Same, body is converted to JSON automatically, but also includes status & headers |
| **Use Case** | Simple API responses, e.g., key-value JSON | When you need **full control** over response: headers, status, body |

1. Returning HashMap



2. Return ResponseEntity



Status code = 201

Header X-Custom-Header: HeaderValue is sent

Full control over response

✅ **In short:**

**HashMap = just data**  
**ResponseEntity = data + headers + status**

**ORM vs Hibernate vs JPA**

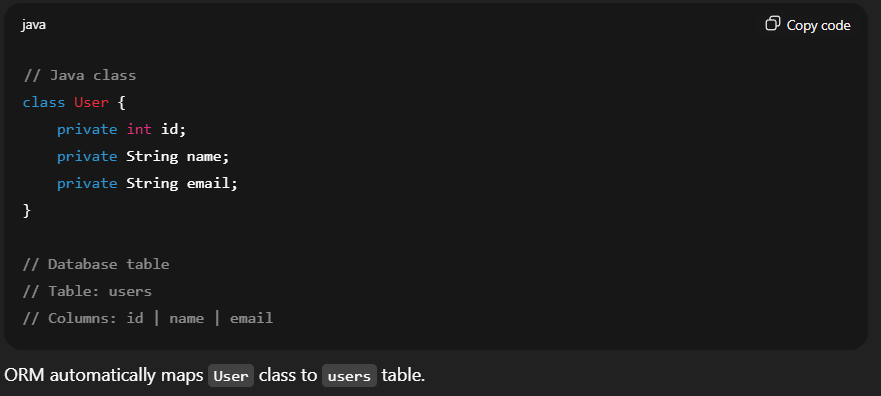
**1. ORM (Object-Relational Mapping)**

**Definition:**

ORM is a **programming technique** that allows you to **map Java objects to database tables** and vice versa.

**Why use ORM?**

* Avoid writing repetitive SQL queries.
* Work with objects instead of raw database tables.
* Maintain **object-oriented programming style**.



ORM automatically maps User class to users table.

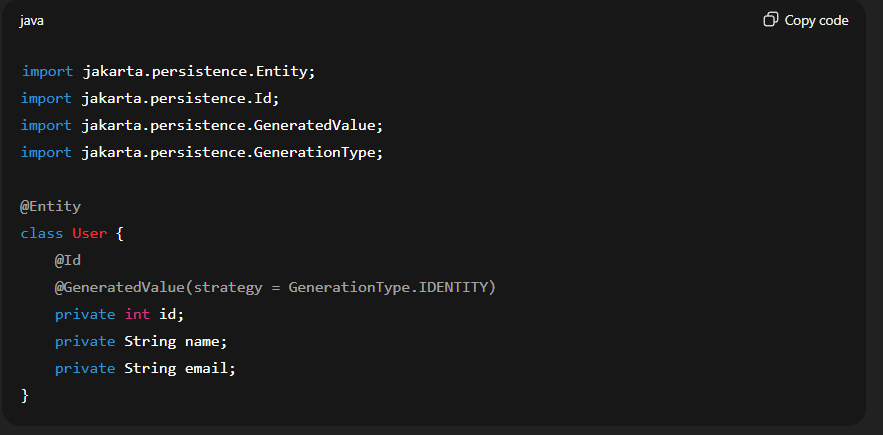
**2. JPA (Java Persistence API)**

**Definition:**

JPA is a **specification (interface)** in Java for ORM. It defines **rules, interfaces, and annotations** for mapping objects to relational databases.

**Key Points:**

* JPA is **not a library** — it’s a standard specification.
* You need an **implementation** to actually persist data.
* Common annotations: @Entity, @Id, @Column, @GeneratedValue.



**3. Hibernate**

**Definition:**

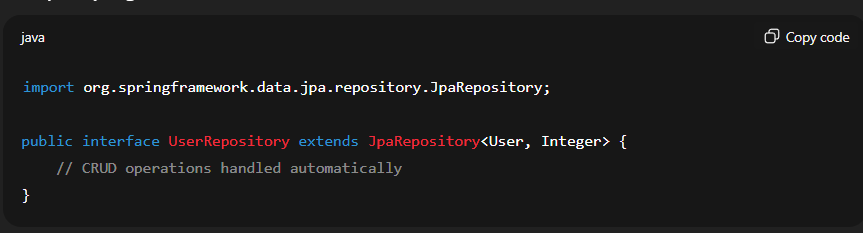
Hibernate is a **popular implementation of JPA**.

* Provides the actual **ORM functionality**, i.e., it implements the JPA interfaces and handles database interactions.

**Key Points:**

* Hibernate supports **JPA annotations** and adds extra features like caching, lazy loading, HQL (Hibernate Query Language).
* You can use Hibernate **with or without JPA**, but in Spring Boot, it’s usually used as a **JPA implementation**.

**Example (Spring Boot with JPA + Hibernate):**



Here:

* UserRepository uses **JPA repository interface** (JPA spec)
* Hibernate (implementation) actually performs the **SQL queries** under the hood.

**Summary Table**

| **Concept** | **Type** | **Role** | **Notes** |
| --- | --- | --- | --- |
| ORM | Concept/Technique | Maps Java objects to DB tables | Avoid raw SQL |
| JPA | Specification/Interface | Defines standard API for ORM | Needs an implementation |
| Hibernate | Library / Framework | Implements JPA | Provides ORM functionality + extra features |

✅ **In short:**

* **ORM** → Concept (mapping objects to DB)
* **JPA** → Standard API (interface/spec)
* **Hibernate** → Actual implementation of JPA (library)

**How Hibernate Works Under the Hood**

1. **JPA provides the specification**
   * You write your entities (@Entity) and repositories (JpaRepository).
   * These are just **interfaces and annotations** — they do not actually talk to the database.
2. **Hibernate is the implementation**
   * Hibernate **reads your JPA entities** and **translates your operations** into SQL.
   * For example:
     + save() → INSERT INTO users ...
     + findById() → SELECT \* FROM users WHERE id = ?
3. **Spring Boot auto-configures everything**
   * spring-boot-starter-data-jpa detects **Hibernate on the classpath**.
   * Creates an **EntityManagerFactory** and **TransactionManager** automatically.
   * So your repositories just work — no manual Hibernate setup needed.

**In short:**

 **JPA dependency** → gives you interfaces & annotations (standard API)

 **Hibernate** → automatically included, implements JPA, talks to the database

 Spring Boot → auto-configures Hibernate → you just use JpaRepository

✅ **In short:**

* **No need to install Hibernate manually** in Spring Boot.
* Spring Boot starter automatically includes it and configures everything for you.

**Workflow Summary (Flow)**

**Frontend → HTTP Request → DispatcherServlet → Controller → Service → Repository → DB → Repository → Service → Controller → JSON Response → Frontend**

* **Key Components:**
  + **DispatcherServlet** – Determines which Controller method should handle the request using HandlerMapping.
  + **Controller** – handles HTTP endpoints
  + **Service** – business logic
  + **Repository** – DB operations
  + **Jackson** – converts objects to JSON