# 1\_Advantages of SpringMVC over Servlet

### **Advantages of Spring MVC over Servlets**

#### **1. Separation of Concerns (SoC):**

* **Servlets:**
  + Business logic, request handling, and response generation are typically written in a single doGet or doPost method.
  + This leads to monolithic code that is hard to maintain.
* **Spring MVC:**
  + Follows the Model-View-Controller (MVC) design pattern.
    - **Model**: Encapsulates application data.
    - **View**: Defines the user interface (JSP, Thymeleaf, etc.).
    - **Controller**: Handles requests and returns appropriate responses.
  + This separation makes the code modular, maintainable, and testable.

#### **2. Annotation-Based Configuration:**

* **Servlets:**
  + Use configuration-heavy web.xml for mapping servlets, filters, and listeners.

Example:  
  
<servlet>

<servlet-name>MyServlet</servlet-name>

<servlet-class>com.example.MyServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>MyServlet</servlet-name>

<url-pattern>/example</url-pattern>

</servlet-mapping>

* **Spring MVC:**
  + Removes the need for web.xml by using annotations like @Controller, @RequestMapping, @GetMapping, etc.

Example:  
  
@Controller

public class MyController {

@GetMapping("/example")

public String handleRequest() {

return "exampleView";

}

}

* + This reduces boilerplate code and makes the application easier to configure.

#### **3. Removal of web.xml (Servlet Configuration):**

* **How Spring MVC Removes web.xml:**
  + With Spring MVC and the introduction of **Servlet 3.0**, configuration is annotation-driven.
  + Use @Configuration and @ComponentScan in a Java-based configuration class to replace web.xml.

Example:  
  
@Configuration

@EnableWebMvc

@ComponentScan(basePackages = "com.example")

public class WebConfig implements WebMvcConfigurer {

@Bean

public ViewResolver viewResolver() {

InternalResourceViewResolver resolver = new InternalResourceViewResolver();

resolver.setPrefix("/WEB-INF/views/");

resolver.setSuffix(".jsp");

return resolver;

}

}

**4. Simplified Request Mapping:**

* **Servlets:**
  + Require explicit handling of HTTP methods and URL patterns.

Example:  
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

if ("/example".equals(request.getPathInfo())) {

response.getWriter().write("Hello World");

}

}

* **Spring MVC:**
  + Simplifies request mapping with annotations like @RequestMapping, @GetMapping, @PostMapping, etc.

Example:  
@GetMapping("/example")

@ResponseBody

public String handleRequest() {

return "Hello World";

}

#### **5. Better Exception Handling:**

* **Servlets:**
  + Developers must write extensive try-catch blocks and handle exceptions manually.
* **Spring MVC:**
  + Provides centralized exception handling with @ExceptionHandler and @ControllerAdvice.

Example:  
  
@ControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(Exception.class)

public String handleException(Exception e) {

return "errorView";

}

}

**6. Dependency Injection:**

* **Servlets:**
  + Dependencies must be instantiated manually, leading to tight coupling.
* **Spring MVC:**
  + Uses Spring's powerful dependency injection mechanism to inject services and DAOs into controllers.

Example:  
  
@Controller

public class MyController {

@Autowired

private MyService myService;

@GetMapping("/example")

public String handleRequest() {

return myService.processRequest();

}

}

#### **7. Eliminating if-else or switch for Request Mapping:**

* **Servlets:**
  + Developers often write lengthy if-else or switch statements to handle different URL patterns.

Example:  
  
String action = request.getParameter("action");

if ("add".equals(action)) {

// Add logic

} else if ("delete".equals(action)) {

// Delete logic

}

* **Spring MVC:**
  + Simplifies this with method-specific mappings.

Example:  
  
@GetMapping("/add")

public String addItem() {

return "addItemView";

}

@GetMapping("/delete")

public String deleteItem() {

return "deleteItemView";

}

#### **8. Integration with View Technologies:**

* **Servlets:**
  + Requires manual forwarding to JSPs or handling response content.
* **Spring MVC:**
  + Provides integration with various view technologies (JSP, Thymeleaf, Freemarker).

Example:  
  
@Controller

public class MyController {

@GetMapping("/example")

public String handleRequest(Model model) {

model.addAttribute("message", "Hello World");

return "exampleView";

}

}

#### **9. Interceptors and Filters:**

* **Servlets:**
  + Require manual configuration for filters and listeners.
* **Spring MVC:**
  + Provides a declarative way to implement cross-cutting concerns like logging and authentication using HandlerInterceptor.

Example:  
java  
Copy code  
public class MyInterceptor implements HandlerInterceptor {

@Override

public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) {

// Logging logic

return true;

}

}

### **Conclusion**

Spring MVC provides a modern, modular, and annotation-driven approach compared to traditional servlets. It removes boilerplate code (web.xml), simplifies request handling, improves testability, and adheres to best practices like separation of concerns and dependency injection. This makes it a robust framework for building scalable web applications.

### **Without Dependency Injection**

public class Payment {

User sender = new User(); // Tight coupling

void getSenderDetails(String id) {

sender.getUserDetails(id); // Direct dependency on the User class

}

}

**Tight Coupling Issue:**

1. **Hardcoded Dependency:**
   * The Payment class explicitly creates an instance of the User class (new User()).
   * This means Payment is tightly tied to the specific implementation of the User class.
2. **Difficult to Mock for Unit Testing:**
   * Since Payment creates a User object directly, you can't inject a mock or fake implementation for testing purposes.
   * Calling getSenderDetails will always invoke the real User class method.
3. **Flexibility Issue:**
   * If the User class evolves (e.g., different types of users like Admin, Member), you’ll need to modify the Payment class code every time you add or change the User implementation.
   * This violates the **Open-Closed Principle (OCP)** of SOLID principles, where a class should be open to extension but closed to modification.

### **With Dependency Injection**

public class Payment {

@Autowired

private User sender; // Injected by the Spring IoC container

void getSenderDetails(String id) {

sender.getUserDetails(id); // Dependency injected at runtime

}

}

**How Dependency Injection Solves the Problems:**

1. **Inversion of Control (IoC):**
   * The creation and management of the User object is delegated to the **Spring IoC container**, not the Payment class.
   * The Payment class only depends on an **interface or abstraction**, not the implementation.
2. **Mocking Made Easy:**
   * For unit testing, you can mock the User dependency and inject it into the Payment class without relying on the real User class.

Example (using Mockito):  
  
@Mock

private User mockUser;

@InjectMocks

private Payment payment;

@Test

public void testGetSenderDetails() {

when(mockUser.getUserDetails(anyString())).thenReturn(mockedData);

payment.getSenderDetails("123");

verify(mockUser).getUserDetails("123");

}

1. **Dynamic User Injection:**
   * You can define multiple User implementations (e.g., Admin, Member) and inject the appropriate one at runtime using Spring features like @Qualifier or profiles.

Example:  
  
@Autowired

@Qualifier("adminUser")

private User sender;

1. **Extensibility and OCP Compliance:**
   * Adding new types of User does not require modifying the Payment class.
   * Instead, you register new beans in the IoC container and use them dynamically.

### **How IoC Helps**

* **IoC** shifts the responsibility of creating and managing dependencies from the Payment class to the **Spring IoC container**.
* This decouples the Payment class from the User implementation and makes the system modular, testable, and extensible.

### **Key Takeaways**

1. **Without DI:**
   * The Payment class directly controls the User object.
   * This leads to tight coupling, poor testability, and difficulty in future changes.
2. **With DI:**
   * The Payment class depends on an abstraction (User bean).
   * Spring IoC container manages the lifecycle, improving flexibility, testability, and adherence to SOLID principles.

# 2\_SpringMVC Setup

Spring MVC setup, along with their purpose and need:

### **1. Controller Class: PaymentController**

@Controller

@RequestMapping("/paymentapi")

public class PaymentController {

@Autowired

PaymentDAO paymentService;

@GetMapping("/payment")

public String getPaymentDetails() {

return paymentService.getDetails();

}

}

#### **Explanation:**

* **@Controller:**
  + Marks this class as a Spring MVC controller.
  + It handles HTTP requests and returns appropriate responses.
* **@RequestMapping("/paymentapi"):**
  + Maps all incoming requests with the base URL /paymentapi to this controller.
  + For example, a request to /paymentapi/payment will be handled by the getPaymentDetails() method.
* **@Autowired:**
  + Injects the PaymentDAO bean into this controller.
  + This allows PaymentController to delegate business logic to the PaymentDAO service.
* **@GetMapping("/payment"):**
  + Maps GET requests to /paymentapi/payment to the getPaymentDetails() method.
* **Method Logic:**
  + The getPaymentDetails() method calls the getDetails() method from the PaymentDAO service and returns the result.

#### **Need:**

The controller serves as the entry point for handling web requests and delegating business logic to the appropriate service layer.

### **2. POM File: pom.xml**

<modelVersion>4.0.0</modelVersion>

<groupId>com.conceptandcoding</groupId>

<artifactId>learningspringboot</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>springboot application</name>

<description>project for learning springboot</description>

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>6.1.4</version>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>servlet-api</artifactId>

<version>2.5</version>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.13.2</version>

<scope>test</scope>

</dependency>

</dependencies>

#### **Explanation:**

* **<modelVersion>:**
  + Specifies the Maven POM version.
  + Version 4.0.0 is the default and widely used.
* **<groupId> and <artifactId>:**
  + Define the unique identifier for your project.
  + com.conceptandcoding.learningspringboot acts as the package structure.
* **<dependencies>:**
  + **spring-webmvc:**
    - Provides Spring MVC support for building web applications.
    - Includes classes for controllers, annotations, and view handling.
  + **servlet-api:**
    - Includes servlet libraries for handling HTTP requests and responses.
  + **junit:**
    - A testing framework for writing and running test cases.

#### **Need:**

The pom.xml file manages dependencies and the build lifecycle of your project using Maven.

### **3. Configuration Class: AppConfig**

@Configuration

@EnableWebMvc

@ComponentScan(basePackages = "com.conceptandcoding")

public class AppConfig {

// Additional configuration if required

}

#### **Explanation:**

* **@Configuration:**
  + Indicates that this class contains configuration settings for Spring.
  + Replaces the traditional applicationContext.xml file.
* **@EnableWebMvc:**
  + Enables Spring MVC features like @Controller, @RequestMapping, and view resolvers.
* **@ComponentScan(basePackages = "com.conceptandcoding"):**
  + Scans the specified package for Spring-managed beans (e.g., controllers, services).

#### **Need:**

The AppConfig class initializes and configures the Spring context, enabling annotations and managing the application's component lifecycle.

### **4. Dispatcher Servlet Class: MyApplicationInitializer**

public class MyApplicationInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

@Override

protected Class<?>[] getRootConfigClasses() {

return null;

}

@Override

protected Class<?>[] getServletConfigClasses() {

return new Class[]{AppConfig.class};

}

@Override

protected String[] getServletMappings() {

return new String[]{"/"};

}

}

#### **Explanation:**

* **AbstractAnnotationConfigDispatcherServletInitializer:**
  + A base class that sets up Spring’s DispatcherServlet and the application context programmatically.
  + Replaces the traditional web.xml.
* **getRootConfigClasses():**
  + Returns the configuration classes for the root application context (e.g., security, persistence).
  + Here, it returns null as there is no root context.
* **getServletConfigClasses():**
  + Returns the configuration classes for the servlet-specific application context.
  + Specifies AppConfig for Spring MVC setup.
* **getServletMappings():**
  + Maps the DispatcherServlet to handle all requests (/).

#### **Need:**

This class bootstraps the Spring MVC application by configuring the DispatcherServlet and linking it to the AppConfig.

### **How the Files Work Together**

1. **POM File:**
   * Brings in required libraries like Spring MVC and servlet APIs.
2. **Dispatcher Servlet (MyApplicationInitializer):**
   * Configures Spring MVC by linking AppConfig and maps the DispatcherServlet.
3. **Configuration Class (AppConfig):**
   * Configures components like controllers and view resolvers.
4. **Controller (PaymentController):**
   * Handles HTTP requests and delegates business logic to the service layer.

### **Request Lifecycle**

1. A request to /paymentapi/payment is received by the DispatcherServlet.
2. The DispatcherServlet uses AppConfig to scan for components.
3. The PaymentController is identified and its getPaymentDetails method is invoked.
4. The response is returned to the client.

This setup enables a clean, modular, and annotation-driven Spring MVC application.

# 3\_Advantages Of SpringBoot over MVC

### **Spring Boot Advantages Over Traditional Spring MVC**

Spring Boot simplifies the development of Spring-based applications by addressing the challenges present in traditional Spring MVC. Here are the key challenges and how Spring Boot resolves them, along with proper notes and explanations:

### **1. Dependency Management**

#### **Challenge:**

* In traditional Spring MVC, developers must manually add and manage multiple dependencies in the pom.xml file.
* Ensuring compatibility between dependencies can be tedious.

#### **Spring Boot Solution:**

* Spring Boot uses a **parent POM** that manages dependencies and their versions automatically.
* This eliminates the need to manage dependency versions manually.

#### **Example POM File:**

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>3.2.3</version>

<relativePath/> <!-- Use parent from the repository -->

</parent>

<groupId>com.conceptandcoding</groupId>

<artifactId>learningspringboot</artifactId>

<version>0.0.1-SNAPSHOT</version>

<properties>

<java.version>17</java.version>

</properties>

<dependencies>

<!-- Web Starter: Includes Spring MVC and Tomcat -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<!-- Test Starter: Includes JUnit and MockMvc -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

#### **Explanation:**

* **spring-boot-starter-parent:** Provides default versions for Spring Boot dependencies.
* **spring-boot-starter-web:** A pre-configured dependency that includes Spring MVC, embedded Tomcat, and other web components.
* **Benefits:**
  + No need to manually manage dependency versions.
  + Ensures compatibility between libraries.

### **2. Auto Configuration**

#### **Challenge:**

* Traditional Spring MVC requires manual configuration of:
  + DispatcherServlet
  + Application context (AppConfig)
  + Annotations like @EnableWebMvc and @ComponentScan.

#### **Spring Boot Solution:**

* Spring Boot provides **auto-configuration**, which automatically sets up necessary components based on the added dependencies.

#### **Key Annotation:**

@SpringBootApplication

public class SpringbootApplication {

public static void main(String[] args) {

SpringApplication.run(SpringbootApplication.class, args);

}

}

#### **Explanation:**

* **@SpringBootApplication:**
  + Combines the functionality of @Configuration, @EnableAutoConfiguration, and @ComponentScan.
  + Scans for Spring-managed components, configures beans, and initializes the application context automatically.
* **Benefits:**
  + No need for a separate configuration class (AppConfig).
  + No need for explicit DispatcherServlet setup.

### **3. Embedded Server**

#### **Challenge:**

* In traditional Spring MVC:
  + Applications are packaged as WAR files.
  + WAR files must be deployed to an external servlet container (e.g., Tomcat).
  + This involves additional steps like deployment and server setup.

#### **Spring Boot Solution:**

* Spring Boot includes an **embedded servlet container** (like Tomcat or Jetty).
* Applications are packaged as JAR files and can be run directly.

#### **Controller Example:**

@RestController

@RequestMapping("/myapi")

public class MyController {

@GetMapping("/firstapi")

public String getData() {

return "Hello from concept and coding";

}

}

#### **How It Works:**

* The spring-boot-starter-web dependency includes an embedded Tomcat server.
* Running the application starts the embedded server, and the application is accessible at http://localhost:8080.

#### **Benefits:**

* Simplifies development and deployment.
* No need to manually set up and configure an external servlet container.
* Supports rapid development and testing.

### **Request Lifecycle in Spring Boot**

1. **Request Mapping:**
   * The user sends a request to http://localhost:8080/myapi/firstapi.
2. **Controller Invocation:**
   * MyController handles the request via the getData() method.
3. **Response Generation:**
   * The method returns a response ("Hello from concept and coding"), which is sent back to the client.

### 

### 

### 

### 

### 

### 

### **Comparison: Spring Boot vs Spring MVC**

| **Feature** | **Spring MVC** | **Spring Boot** |
| --- | --- | --- |
| **Dependency Management** | Manually manage dependencies and versions. | Uses spring-boot-starter-parent for simplified dependency management. |
| **Configuration** | Requires manual configuration (e.g., AppConfig, DispatcherServlet). | Auto-configures components based on dependencies. |
| **Server Setup** | Requires external servlet container (WAR deployment). | Includes an embedded server (Tomcat, Jetty, etc.). |
| **Startup** | Requires additional setup for server and deployment. | Directly runs with SpringApplication.run(). |
| **Ease of Use** | More boilerplate code. | Simplified setup and faster development. |

### **Key Takeaways**

1. **Dependency Management:**
   * Spring Boot handles dependency versions and compatibility.
2. **Auto Configuration:**
   * Reduces boilerplate code by auto-configuring required components.
3. **Embedded Server:**
   * Eliminates the need for WAR deployment, allowing rapid development and testing.

Spring Boot makes application development faster, easier, and less error-prone compared to traditional Spring MVC.

# 4\_SpringBoot Architecture

### **Flow Explanation with Visualization**

The provided code demonstrates a structured implementation of a **Payment API** in Spring Boot, following a layered architecture: **Controller → Service → Repository**. Each layer has a specific responsibility, ensuring separation of concerns.

#### **Flow Overview**

1. **Client Request:**
   * The client sends a GET request to the endpoint /payments/{id}.
   * Example: GET http://localhost:8080/payments/1
2. **Controller Layer:**
   * PaymentController handles the HTTP request and maps it to a PaymentRequest DTO.
   * The request is forwarded to the PaymentService.
3. **Service Layer:**
   * PaymentService receives the PaymentRequest and interacts with the PaymentRepository to fetch data.
   * Maps the PaymentEntity retrieved from the database to a PaymentResponse DTO.
4. **Repository Layer:**
   * PaymentRepository fetches the data from the database.
   * A mock database query (executeQuery) returns a PaymentEntity.
5. **Response to Client:**
   * The PaymentResponse DTO is returned as the HTTP response.

### **Visualization**

#### **1. Client Request**

* **Request:** GET http://localhost:8080/payments/1
* The client specifies the paymentId in the URL.

#### **2. Controller Layer:**

* **File:** PaymentController.java

**Responsibilities:**

* Accepts the HTTP request.
* Maps the incoming request (id) to a PaymentRequest DTO.
* Passes the DTO to the service layer.

**Flow:**

1. Extracts id from the URL.
2. Maps it to a PaymentRequest DTO.
3. Calls paymentService.getPaymentsDetailsById(internalReqObj).

#### **3. Service Layer:**

* **File:** PaymentService.java

**Responsibilities:**

* Orchestrates the business logic.
* Interacts with the repository to fetch data.
* Maps PaymentEntity to PaymentResponse.

**Flow:**

1. Calls paymentRepository.getPaymentById(internalReqObj).
2. Receives a PaymentEntity from the repository.
3. Maps the PaymentEntity to a PaymentResponse DTO.
4. Returns the PaymentResponse to the controller.

#### **4. Repository Layer:**

* **File:** PaymentRepository.java

**Responsibilities:**

* Connects to the database.
* Fetches the PaymentEntity based on the PaymentRequest.

**Flow:**

1. Executes a mock query (executeQuery(request)).
2. Creates a PaymentEntity object with dummy data.
   * paymentId: Fetched from PaymentRequest.
   * paymentCurrency: Hardcoded as "INR".
   * paymentAmount: Hardcoded as 100.

#### **5. Response to Client**

* **DTO Returned:** PaymentResponse

**Content:**

* paymentId: 1
* amount: 100.0
* currency: "INR"

### **Diagram**

Client

↓

HTTP GET /payments/1

↓

PaymentController

- Maps URL `id` to `PaymentRequest`

- Calls PaymentService

↓

PaymentService

- Calls PaymentRepository

- Maps PaymentEntity → PaymentResponse

↓

PaymentRepository

- Executes Query

- Returns PaymentEntity

↓

Database (Mocked)

- Returns data for paymentId = 1

↓

PaymentService

- Converts PaymentEntity → PaymentResponse

↓

PaymentController

- Returns PaymentResponse

↓

Client

- Receives Response:

{

"paymentId": 1,

"amount": 100.0,

"currency": "INR"

}

### **Code Layer Breakdown**

1. **Controller Layer:**
   * **Purpose:** Handles the HTTP request and delegates to the service layer.

**Key Code:**  
@GetMapping("/{id}")

public ResponseEntity<PaymentResponse> getPaymentById(@PathVariable Long id) {

PaymentRequest internalReqObj = new PaymentRequest();

internalReqObj.setPaymentId(id);

PaymentResponse payment = paymentService.getPaymentsDetailsById(internalReqObj);

return ResponseEntity.ok(payment);

}

1. **Service Layer:**
   * **Purpose:** Implements business logic and communicates with the repository.

**Key Code:**  
public PaymentResponse getPaymentsDetailsById(PaymentRequest internalReqObj) {

PaymentEntity paymentModel = paymentRepository.getPaymentById(internalReqObj);

return mapModelToResponseDTO(paymentModel);

}

private PaymentResponse mapModelToResponseDTO(PaymentEntity paymentEntity) {

PaymentResponse response = new PaymentResponse();

response.setPaymentId(paymentEntity.getId());

response.setAmount(paymentEntity.getPaymentAmount());

response.setCurrency(paymentEntity.getPaymentCurrency());

return response;

}

1. **Repository Layer:**
   * **Purpose:** Fetches data from the database (mocked here).

**Key Code:**  
public PaymentEntity getPaymentById(PaymentRequest request) {

return executeQuery(request);

}

private PaymentEntity executeQuery(PaymentRequest request) {

PaymentEntity payment = new PaymentEntity();

payment.setId(request.getPaymentId());

payment.setPaymentCurrency("INR");

payment.setPaymentAmount(100);

return payment;

}

### **Advantages of This Flow**

1. **Separation of Concerns:**
   * Controller, Service, and Repository have distinct roles.
2. **Reusability:**
   * DTOs and mapping functions can be reused across services.
3. **Scalability:**
   * Easily extendable to include more fields or new endpoints.
4. **Testability:**
   * Each layer can be independently tested.

This structured approach ensures maintainable and efficient code development.

## JAR / WAR

### **WAR and JAR Files: Definition and Purpose**

Both **WAR** (Web Application Archive) and **JAR** (Java Archive) files are packaging formats used in Java development. They are compressed files that bundle resources, class files, and libraries to make Java applications portable and easier to deploy.

### **1. JAR (Java Archive)**

* **What is it?**
  + A **JAR** file is a compressed package containing Java class files, metadata, and other resources like images, properties files, or configuration files. It is primarily used for packaging standalone Java applications or reusable libraries.
* **Why is it needed?**
  + **Reusability:** Developers can package reusable libraries (e.g., logging frameworks, database drivers) into JARs.
  + **Portability:** A JAR file bundles all the components into a single file, simplifying the distribution process.
  + **Execution:** Some JAR files are executable (contain a MANIFEST.MF file with the Main-Class entry) and can be run directly using java -jar.
* **Example Use Cases:**
  + Packaging a desktop Java application.
  + Creating utility libraries for use in multiple projects.

### **2. WAR (Web Application Archive)**

* **What is it?**
  + A **WAR** file is a compressed package designed for deploying web applications on a Java servlet container (e.g., Apache Tomcat, Jetty). It contains:
    - **HTML/JS/CSS files:** Front-end content.
    - **Servlets:** Java class files handling backend logic.
    - **JSPs (Java Server Pages):** Dynamic web pages.
    - **Configuration files:** web.xml, Spring configuration, etc.
    - **Libraries:** Required JAR files.
* **Why is it needed?**
  + **Standardized Deployment:** A WAR file contains all the components of a web application in a predefined structure, making it compatible with servlet containers.
  + **Convenience:** Simplifies deploying, managing, and distributing web applications.
  + **Scalability:** WAR files allow modular updates and can be redeployed without affecting the server.
* **Example Use Cases:**
  + Deploying a Spring MVC application on Apache Tomcat.
  + Hosting a Java-based e-commerce website.

### **Key Differences Between JAR and WAR**

| **Feature** | **JAR** | **WAR** |
| --- | --- | --- |
| **Full Form** | Java Archive | Web Application Archive |
| **Usage** | Standalone Java applications, libraries | Web applications |
| **Contains** | Class files, libraries, resources | Class files, JSP, servlets, resources, configuration |
| **Execution** | Can be executed with java -jar | Deployed on servlet containers |
| **Structure** | No specific structure required | Predefined structure (e.g., WEB-INF, META-INF) |

### 

### 

### 

### 

### 

### 

### 

### 

### 

### **Why Are These Needed?**

1. **Ease of Distribution:**
   * Bundles everything required for an application in one file.
   * Simplifies sharing or deploying applications.
2. **Modularity:**
   * Libraries in JARs can be shared across multiple projects.
   * Web applications in WARs can be independently deployed on servers.
3. **Compatibility:**
   * Ensures standardization for execution in Java environments.
   * WAR files are compatible with all Java EE-compliant servlet containers.
4. **Deployment Convenience:**
   * WAR files simplify deploying web apps on servers.
   * JAR files streamline running standalone apps or importing dependencies.

### **Conclusion**

* **JAR files**: Best for packaging standalone applications or reusable libraries.
* **WAR files**: Ideal for packaging and deploying Java-based web applications on servers. Both formats ensure modularity, portability, and compatibility, making Java development and deployment efficient.

## POJO Classes

### **What is a POJO Class?**

A **POJO** (Plain Old Java Object) class is a simple Java class with:

* **No special constraints or requirements.**
* **No specific inheritance or annotations (unless necessary).**
* Contains:
  + **Fields:** Private variables representing data.
  + **Getters and Setters:** Methods to access and modify fields.
  + **Constructors (optional):** For initializing fields.
  + **No business logic or extra methods.**

### **Characteristics of a POJO Class**

1. **Encapsulation:** Fields are private, and access is provided through getters and setters.
2. **No Framework Dependency:** It doesn’t depend on any frameworks or APIs.
3. **Lightweight:** Used for holding data only, not for implementing complex logic.
4. **Easy Serialization:** Can be serialized if needed (e.g., implementing Serializable).

### **Use of POJO Classes**

In Java applications, POJO classes serve as:

1. **Data Transfer Objects (DTOs):** For passing data between different layers.
2. **Model Objects:** Representing real-world entities (e.g., Users, Products).
3. **Entities:** When annotated with ORM (e.g., JPA) annotations, they map directly to database tables.

### **POJO Classes in Your Example**

#### **Entity Class: PaymentEntity**

In your example, PaymentEntity represents a **direct mapping to the database table.**

* This class typically has fields corresponding to the columns in the table, like Id, paymentAmount, paymentCurrency, and userEmail.
* It's an **Entity class** because it represents the database structure.
* With an ORM framework like JPA, it would usually have annotations like @Entity, @Table, and @Column.

#### **POJO Class: PaymentRequestDTO and PaymentResponseDTO**

* **POJO Classes:**
  + PaymentRequestDTO and PaymentResponseDTO are **DTOs (Data Transfer Objects)** that encapsulate and carry data between layers (e.g., Controller and Service).
  + They **do not directly represent the database table.**
* **Characteristics in Your Example:**
  + **PaymentRequestDTO**: Encapsulates the input request data (e.g., paymentId).
  + **PaymentResponseDTO**: Encapsulates the response data (e.g., paymentId, amount, currency).

### **How Do POJO Classes Differ from Entity Classes?**

| **Aspect** | **Entity Class** | **POJO (DTO) Class** |
| --- | --- | --- |
| **Purpose** | Maps directly to database tables | Transfers data between layers |
| **Annotations** | Usually annotated (@Entity) | No annotations (pure Java object) |
| **Relation to Database** | Represents database structure | Independent of the database |
| **Logic** | No logic (only data persistence) | May include lightweight data transformation logic |
| **Example in Your Code** | PaymentEntity | PaymentRequestDTO, PaymentResponseDTO |

### 

### 

### 

### **Visualizing the Flow in Your Code**

1. **Request:**
   * PaymentRequestDTO is created with incoming data (e.g., paymentId).
2. **Mapping and Processing:**
   * The Service layer fetches data using PaymentEntity (represents the database table).
   * Data is mapped from PaymentEntity to PaymentResponseDTO.
3. **Response:**
   * PaymentResponseDTO is returned as the output to the client.

This separation ensures:

* **Clean architecture:** DTOs are specific to the application layer, while entities deal with the persistence layer.
* **Flexibility:** Changes in database schema (entity) won’t directly impact DTOs.

# 5\_Maven Lifecycle

# **Maven: A Project Management Tool**

Maven is a powerful project management and build automation tool. It uses the Project Object Model (POM), an XML file that defines the project’s dependencies, build lifecycle, and other configurations.

## **Phases in Maven Build Lifecycle**

Maven follows a build lifecycle consisting of several phases. When a specific phase is run, Maven also executes all preceding phases.

### **1. Validate Phase**

* Ensures that the project structure is correct.
* Checks for the presence of necessary directories like src, target, and configuration files.
* Example Goal: Validate that the pom.xml is well-formed and references valid dependencies.

### **2. Compile Phase**

* Compiles the source code located in the src/main/java directory.
* Uses the javac compiler to generate .class files in the target/classes directory.
* Fails if there are compilation errors.

Command:

mvn compile

### **3. Test Phase**

* Runs unit tests using a testing framework like JUnit or TestNG.
* Executes test classes located in the src/test/java directory.
* Does not stop the build even if tests fail (unless explicitly configured).

Command:

mvn test

### **4. Package Phase**

* Generates a deployable package such as a JAR or WAR file.
* The package is stored in the target directory.
* Requires a proper directory structure and pom.xml configurations.

Command:

mvn package

### **5. Verify Phase**

* Verifies the integrity of the package.
* Often used for tasks like:
  + Running static code analysis (e.g., PMD, Checkstyle).
  + Ensuring all tests have passed successfully.
* Example Tools:
  + **PMD:** Finds code issues such as unused variables, duplicate code, and empty catch blocks.

Command:

mvn verify

### **6. Install Phase**

* Installs the generated package (JAR/WAR) into the local Maven repository for local use.

The local repository is typically located at:  
 ~/.m2/repository

Command:

mvn install

### **7. Deploy Phase**

* Deploys the package to a remote repository for sharing with other developers or production use.
* Requires configurations in the pom.xml and settings.xml files for credentials and server details.

#### **Example Configuration in pom.xml:**

<distributionManagement>

<repository>

<id>remote-repo-id</id>

<url>https://remote-repo-url</url>

</repository>

</distributionManagement>

Command:

mvn deploy

## **Key Notes on Maven Phases**

1. **Phases Are Sequential:** Running a phase executes all preceding phases as well. For example, running the package phase will first execute validate, compile, and test.
2. **Each Phase Can Have Multiple Goals:** Specific tasks (goals) associated with each phase are executed in order.
3. **Custom Goals:** You can add custom goals to any phase using plugins in the pom.xml file.

#### **Example:**

<build>

<plugins>

<plugin>

<phase>validate</phase>

<goals>

<goal>check</goal>

</goals>

</plugin>

</plugins>

</build>

## **Key Elements in pom.xml**

### **1. Properties**

* Key-value pairs for configuration.

Example:

<properties>

<java.version>21</java.version>

</properties>

### **2. Repositories**

* Specifies where Maven should download dependencies from.

### **3. Build**

* Configures plugins and additional tasks for various phases.

## **Maven Local Repository**

* Stores downloaded dependencies and built packages for local use.
* Default Location: ~/.m2/repository.

## 

## 

## 

## **Summary**

| **Phase** | **Description** |
| --- | --- |
| Validate | Ensures project structure and configuration are correct. |
| Compile | Compiles the source code. |
| Test | Runs unit tests. |
| Package | Creates a JAR/WAR file in the target directory. |
| Verify | Verifies the package integrity and quality. |
| Install | Installs the package to the local Maven repository for local use. |
| Deploy | Deploys the package to a remote repository for team or production usage. |

# 6\_Annotations

### **Annotations in Spring Boot**

Annotations simplify configuration and development by providing declarative metadata to Spring Boot applications. Below are key Spring Boot annotations with explanations and examples:

### **1. @Controller**

* **Purpose**: Handles incoming HTTP requests.
* **Details**:
  + Methods within a @Controller return views (e.g., JSP or Thymeleaf templates) by default.

Example:

@Controller

public class MyController {

@GetMapping("/greet")

public String greet() {

return "hello"; // Maps to a "hello.jsp" or "hello.html" view.

}

}

### **2. @RestController**

* **Purpose**: Combines @Controller and @ResponseBody.
* **Details**:
  + Suitable for REST APIs.
  + The method return type is directly serialized into the HTTP response body.

Example:  
  
@RestController

public class MyRestController {

@GetMapping("/api/greet")

public String greet() {

return "Hello, World!"; // Returned directly as HTTP response body.

}

}

### **3. @RequestMapping**

* **Purpose**: Maps HTTP requests to controller methods or classes.
* **Attributes**:
  + path: Specifies the URL path.
  + method: Defines HTTP methods like GET, POST, PUT, etc.
  + consumes: Specifies the accepted media types (e.g., JSON).
  + produces: Specifies the response media type.

**Example**:  
  
@RequestMapping(path = "/api/data", method = RequestMethod.GET, produces = "application/json")

public String getData() {

return "{\"key\":\"value\"}";

}

### **4. @RequestParam**

* **Purpose**: Binds HTTP request parameters to method parameters.
* **Details**:
  + Automatically converts query parameters into Java types.

Example:  
  
@GetMapping("/welcome")

public String welcome(@RequestParam("name") String name) {

return "Welcome, " + name;

}

// URL: /welcome?name=John -> Response: Welcome, John

### **5. @PathVariable**

* **Purpose**: Binds URL path variables to method parameters.

**Example**:  
@GetMapping("/user/{id}")

public String getUser(@PathVariable("id") int userId) {

return "User ID: " + userId;

}

// URL: /user/101 -> Response: User ID: 101

### **6. @RequestBody**

* **Purpose**: Binds the HTTP request body to a Java object (typically JSON).
* **Details**:
  + Uses libraries like Jackson or Gson for JSON conversion.

Example:  
  
@PostMapping("/save")

public String saveUser(@RequestBody User user) {

return "User saved: " + user.getName();

}

### **7. @ResponseEntity**

* **Purpose**: Represents the entire HTTP response, including headers, status code, and body.
* **Details**:
  + Provides greater control over the response.

Example:  
  
@GetMapping("/response")

public ResponseEntity<String> response() {

return new ResponseEntity<>("Success", HttpStatus.OK);

}

### **Comparison Between @RequestParam and @PathVariable**

| **Feature** | **@RequestParam** | **@PathVariable** |
| --- | --- | --- |
| **Usage** | Extracts query parameters from the URL. | Extracts values from the URL path. |
| **Example URL** | /api?name=John | /api/John |
| **Declaration Example** | @RequestParam("name") String name | @PathVariable("name") String name |

## Property Editor

### **What is a PropertyEditor?**

* A **PropertyEditor** allows custom conversion or preprocessing of request parameters before they are mapped to method parameters.
* It is often used to handle special cases like trimming whitespace, converting to lowercase, etc.

### **Code Explained**

#### **1. Controller Class**

The controller handles incoming requests, processes the parameters, and uses the custom property editor for specific fields.

java

Copy code

@RestController

@RequestMapping(value = "/api/")

public class SampleController {

// InitBinder to register custom PropertyEditor

@InitBinder

protected void initBinder(WebDataBinder binder) {

// Registers a custom PropertyEditor for the "firstName" field

binder.registerCustomEditor(String.class, "firstName", new FirstNamePropertyEditor());

}

// API endpoint to fetch user details

@GetMapping(path = "/fetchUser")

public String getUserDetails(

@RequestParam(name = "firstName") String firstName,

@RequestParam(name = "lastName", required = false) String lastName,

@RequestParam(name = "age") int age) {

// Logic to return user details

return String.format(

"Fetching and returning user details based on first name: %s, last name: %s, and age: %d",

firstName, lastName, age);

}

}

#### **2. Custom PropertyEditor Class**

The **FirstNamePropertyEditor** class customizes the behavior of how the "firstName" parameter is processed.

java

Copy code

import java.beans.PropertyEditorSupport;

public class FirstNamePropertyEditor extends PropertyEditorSupport {

@Override

public void setAsText(String text) throws IllegalArgumentException {

// Trims whitespace and converts the value to lowercase

setValue(text.trim().toLowerCase());

}

}

### **Flow Explanation**

1. **Request Parameters**:
   * A user sends a request to /api/fetchUser with query parameters like ?firstName= JOHN &lastName=Doe&age=25.
2. **@InitBinder Annotation**:
   * Spring invokes the initBinder method before mapping request parameters to method arguments.
   * The custom FirstNamePropertyEditor is registered for the field firstName.
3. **Custom PropertyEditor Logic**:
   * When Spring processes the firstName parameter, it uses FirstNamePropertyEditor:
     + Trims whitespace from firstName.
     + Converts the value to lowercase.
4. **Processed Method Parameter**:
   * The method getUserDetails will receive:
     + firstName: "john"
     + lastName: "Doe"
     + age: 25.
5. **Response**:

The method returns a response like:  
sql  
Copy code  
Fetching and returning user details based on first name: john, last name: Doe, and age: 25

### **Key Points**

* **Why Use @InitBinder?**
  + It allows pre-processing of specific fields before binding them to method parameters.
  + Especially useful for repetitive tasks like formatting, trimming, or validation.
* **When to Use PropertyEditor?**
  + When you need to preprocess input parameters for specific fields without modifying them manually inside the controller logic.

### **Final Notes**

* The **PropertyEditor** approach is suitable for basic tasks.
* For complex cases (e.g., validating or converting objects), prefer using **@ModelAttribute** or **@Validated** annotations with custom converters or validators.

## Model Attribute & Validated

### **1. Using @ModelAttribute for Binding and Preprocessing**

@ModelAttribute is used to bind HTTP request parameters to an object and preprocess the object before passing it to a controller method. It is particularly useful when you deal with complex objects (e.g., form submissions).

#### **Example: Using @ModelAttribute**

##### **Step 1: Create a DTO Class**

This class represents the incoming request data.

public class UserDetails {

private String firstName;

private String lastName;

private int age;

// Getters and Setters

public String getFirstName() {

return firstName;

}

public void setFirstName(String firstName) {

this.firstName = firstName;

}

public String getLastName() {

return lastName;

}

public void setLastName(String lastName) {

this.lastName = lastName;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

}

##### **Step 2: Define the Controller**

Use @ModelAttribute to bind and preprocess the data.

@RestController

@RequestMapping("/api")

public class SampleController {

@InitBinder

protected void initBinder(WebDataBinder binder) {

// Register custom PropertyEditor for preprocessing

binder.registerCustomEditor(String.class, "firstName", new FirstNamePropertyEditor());

}

@GetMapping("/fetchUser")

public String getUserDetails(@ModelAttribute UserDetails userDetails) {

// Access the preprocessed fields

return String.format("User Details - First Name: %s, Last Name: %s, Age: %d",

userDetails.getFirstName(), userDetails.getLastName(), userDetails.getAge());

}

}

##### 

##### 

##### **Step 3: Request and Response Flow**

**Request URL**:  
  
/api/fetchUser?firstName= JOHN &lastName=Doe&age=25

1. **Processing**:
   * @ModelAttribute binds the query parameters to the UserDetails object.
   * The FirstNamePropertyEditor trims and formats firstName.

**Response**:  
  
User Details - First Name: john, Last Name: Doe, Age: 25

### **2. Using Validation Framework with @Validated**

For more robust validation, Spring Boot supports the **Java Bean Validation API** (JSR 380), using annotations like @NotNull, @Size, etc., on the DTO class.

#### **Example: Using @Validated**

##### **Step 1: Add Validation Annotations**

Update the DTO class with validation constraints.

import jakarta.validation.constraints.NotBlank;

import jakarta.validation.constraints.NotNull;

import jakarta.validation.constraints.Min;

public class UserDetails {

@NotBlank(message = "First name cannot be blank")

private String firstName;

private String lastName;

@NotNull(message = "Age is required")

@Min(value = 1, message = "Age must be greater than 0")

private Integer age;

// Getters and Setters

}

##### **Step 2: Enable Validation in the Controller**

Use @Validated to trigger validation and handle errors with BindingResult.

import org.springframework.validation.BindingResult;

import org.springframework.validation.annotation.Validated;

@RestController

@RequestMapping("/api")

public class SampleController {

@InitBinder

protected void initBinder(WebDataBinder binder) {

binder.registerCustomEditor(String.class, "firstName", new FirstNamePropertyEditor());

}

@PostMapping("/fetchUser")

public String getUserDetails(@Validated @ModelAttribute UserDetails userDetails, BindingResult result) {

if (result.hasErrors()) {

return result.getAllErrors().toString(); // Return validation errors

}

return String.format("User Details - First Name: %s, Last Name: %s, Age: %d",

userDetails.getFirstName(), userDetails.getLastName(), userDetails.getAge());

}

}

##### 

##### 

##### **Step 3: Request and Response Flow**

**Valid Request**:

POST /api/fetchUser

Body: firstName= JOHN &lastName=Doe&age=25

Response:

User Details - First Name: john, Last Name: Doe, Age: 25

**Invalid Request**:

POST /api/fetchUser

Body: firstName= &age=-1

Response:

[ Field error in object 'userDetails' on field 'firstName': rejected value [];

codes [NotBlank.userDetails.firstName,NotBlank.firstName,NotBlank.java.lang.String,NotBlank];

arguments [org.springframework.context.support.DefaultMessageSourceResolvable: codes [userDetails.firstName,firstName]; arguments []; default message [firstName]];

default message [First name cannot be blank],

Field error in object 'userDetails' on field 'age': rejected value [-1];

codes [Min.userDetails.age,Min.age,Min.java.lang.Integer,Min]; arguments

[1,org.springframework.context.support.DefaultMessageSourceResolvable: codes [userDetails.age,age]; arguments []; default message [age]];

default message [Age must be greater than 0]

]

### **Comparison of Approaches**

| **Feature** | **@ModelAttribute** | **@Validated** |
| --- | --- | --- |
| **Use Case** | Basic preprocessing and data binding. | Advanced validation and error handling. |
| **Customization** | Can use PropertyEditor for field-level tweaks. | Uses annotations for validation rules. |
| **Error Handling** | Requires manual checks for invalid data. | Automatic error messages with BindingResult. |
| **Complex Objects** | Best suited for binding simple or complex objects. | Ideal for validating complex objects. |

### **Conclusion**

* Use **@ModelAttribute** when you want to bind and preprocess request data.
* Use **@Validated** when you want built-in validation with minimal effort and error handling.

## Request Body

### **@RequestBody and JSON Binding Process**

In the provided example, the @RequestBody annotation is used in the SampleController to bind the incoming HTTP request body (in JSON format) to a Java object (User). Here’s how this works step by step:

### **1. The Controller Code**

@RestController

@RequestMapping(value = "/api/")

public class SampleController {

@PostMapping(path = "/saveUser")

public String getUserDetails(@RequestBody User user) {

return "User created " + user.username + ":" + user.email;

}

}

#### **Key Points:**

* **@RestController**: Indicates that this class is a REST controller where all methods return data, not views.
* **@PostMapping**: Maps HTTP POST requests to the getUserDetails method.
* **@RequestBody**: Binds the JSON request body to the User object.

### **2. The User Class**

public class User {

@JsonProperty("user\_name")

String username;

String email;

// Getters and Setters

public String getUsername() {

return username;

}

public void setUsername(String username) {

this.username = username;

}

public String getEmail() {

return email;

}

public void setEmail(String email) {

this.email = email;

}

}

#### **Key Points:**

* **@JsonProperty("user\_name")**: This maps the JSON field "user\_name" to the username property of the User class. This is particularly useful if the JSON keys differ from your Java field names.
* Getters and setters are necessary for serialization and deserialization by libraries like **Jackson**.

### **3. The CURL Command**

curl --location --request POST 'http://localhost:8080/api/saveUser' \

--header 'Content-Type: application/json' \

--data-raw '{

"user\_name": "Shrayansh",

"email": "sjxyztest@gmail.com"

}'

#### **Key Points:**

* **URL**: http://localhost:8080/api/saveUser sends the POST request to the saveUser endpoint.
* **Header**: Content-Type: application/json specifies that the request body is in JSON format.

**Body**:  
{

"user\_name": "Shrayansh",

"email": "sjxyztest@gmail.com"

}

* This is the JSON payload sent in the request.

### **4. How the Request Body is Processed**

#### **Step 1: JSON to Java Object Mapping**

The JSON payload is:  
{

"user\_name": "Shrayansh",

"email": "sjxyztest@gmail.com"

}

* Spring uses the **Jackson library** (by default) to deserialize this JSON into a User object:
  + user\_name maps to username because of the @JsonProperty annotation.
  + email maps directly to the email field.

#### **Step 2: Java Object in the Controller**

After deserialization, the User object in the getUserDetails method will have the following state:

User user = new User();

user.setUsername("Shrayansh");

user.setEmail("sjxyztest@gmail.com");

#### 

#### 

#### **Step 3: Response Generation**

The controller method generates the response using the User object:

return "User created Shrayansh:sjxyztest@gmail.com";

The response sent to the client will be:

User created Shrayansh:sjxyztest@gmail.com

### **5. Summary of the @RequestBody Workflow**

1. **JSON Request Body**: The client sends a JSON payload in the request body.
2. **Deserialization**: The @RequestBody annotation triggers deserialization of the JSON into the specified Java object (User).
3. **Controller Logic**: The controller processes the Java object as needed.
4. **Response**: The controller sends a response back to the client.

### **Common Issues with @RequestBody**

1. **Missing or Incorrect JSON Fields**:
   * If user\_name or email is missing, deserialization may fail or result in null fields in the User object.
   * Ensure that the JSON payload matches the expected structure.
2. **Content-Type Mismatch**:
   * If the Content-Type header is not set to application/json, Spring may not process the request body correctly.
3. **Case Sensitivity**:
   * Field names in JSON must match the @JsonProperty values or Java field names (if @JsonProperty is not used).
4. **Validation**:
   * You can use validation annotations like @NotNull or @Size on the User class fields to enforce constraints.

### 

### **Example with Validation**

Update the User class with validation annotations:

import jakarta.validation.constraints.NotBlank;

public class User {

@JsonProperty("user\_name")

@NotBlank(message = "Username is required")

String username;

@NotBlank(message = "Email is required")

String email;

// Getters and Setters

}

Update the controller to handle validation:

@PostMapping(path = "/saveUser")

public String getUserDetails(@Valid @RequestBody User user) {

return "User created " + user.getUsername() + ":" + user.getEmail();

}

If validation fails, Spring automatically returns an error response with the validation messages.

# 7\_Bean Lifecycle

### **Bean Lifecycle in Spring Framework**

The **Spring Bean Lifecycle** defines the process of how a Spring IoC (Inversion of Control) container creates, initializes, and manages beans during the application runtime. The lifecycle consists of several stages, and specific annotations or interfaces can be used to hook into these stages.

### **Key Stages of the Spring Bean Lifecycle**

#### **1. Application Context Initialization**

* When the application starts, the IoC container is initialized with the configuration (e.g., via @ComponentScan).
* The container scans for classes annotated with @Component, @Service, @Repository, or @Controller to identify beans to create.

#### **2. Bean Creation**

* The container creates instances of the identified classes (beans) using their constructors.
* If a bean is marked with @Lazy, it is not created immediately but only when it is first accessed.

##### **Example Flow:**

@Component

public class User {

@Autowired

Order order; // Dependency to be injected.

public User() {

System.out.println("User Created");

}

}

@Lazy

@Component

public class Order {

public Order() {

System.out.println("Order Created");

}

}

When the application starts:

1. The User bean is created immediately since it is not marked as @Lazy.
   * Console Output: User Created.
2. The Order bean is marked as @Lazy, so it is created only when accessed.

#### **3. Dependency Injection**

* After creating the bean, Spring resolves its dependencies using:
  + **Constructor Injection**
  + **Field Injection** (e.g., @Autowired)
  + **Setter Injection**

##### **Example:**

* In the User class, the Order bean is injected into the User bean:

@Autowired

Order order;

Spring looks for a bean of type Order in the container and injects it.

#### **4. Post-Construction (@PostConstruct)**

* Once dependencies are injected, Spring invokes the method annotated with @PostConstruct (if present).
* This method can be used to perform any initialization tasks before the bean is ready for use.

##### 

##### **Example:**

@Component

public class User {

@PostConstruct

public void init() {

System.out.println("User Bean is ready for use");

}

}

Console Output (after dependency injection):

User Bean is ready for use

#### **5. Bean Usage**

* The bean is now fully initialized and ready to be used for executing business logic.
* Application code can invoke methods on the bean.

#### **6. Pre-Destruction (@PreDestroy)**

* When the application context shuts down, Spring calls the method annotated with @PreDestroy (if present) to perform cleanup tasks.
* This is the last stage before the bean is destroyed.

##### **Example:**

@Component

public class User {

@PreDestroy

public void destroy() {

System.out.println("User Bean is being destroyed");

}

}

Console Output (on shutdown):

User Bean is being destroyed

### **Annotated Lifecycle Stages with Example Flow**

Here is how the lifecycle applies to your example:

@Component

public class User {

@Autowired

Order order;

public User() {

System.out.println("User Created");

}

@PostConstruct

public void init() {

System.out.println("User Bean is ready for use");

}

@PreDestroy

public void destroy() {

System.out.println("User Bean is being destroyed");

}

}

@Lazy

@Component

public class Order {

public Order() {

System.out.println("Order Created");

}

}

#### 

#### **Application Startup Flow:**

1. **Bean Discovery**: The @ComponentScan detects User and Order as beans.
2. **Bean Creation**:
   * User bean is created immediately (User Created printed).
   * Order bean is lazy, so it’s not created yet.
3. **Dependency Injection**:
   * The Order bean is injected into the User bean. If accessed here, Order Created is printed.
4. **Post-Construction**:
   * The @PostConstruct method (init()) is invoked on the User bean.

Console Output:  
User Created

Order Created

User Bean is ready for use

#### **During Bean Usage:**

* Methods in the User or Order beans can be invoked to perform business logic.

#### **Application Shutdown Flow:**

1. **Pre-Destruction**:
   * The @PreDestroy method (destroy()) is called on the User bean during context shutdown.

Console Output:  
User Bean is being destroyed

### 

### 

### 

### 

### **Key Annotations and Methods in Bean Lifecycle**

| **Stage** | **Purpose** | **Annotation/Method** | **Example** |
| --- | --- | --- | --- |
| **Bean Discovery** | Identifies classes to be managed as beans | @ComponentScan | @ComponentScan("com.example") |
| **Bean Creation** | Creates the instance of the bean | Constructor | public User() {} |
| **Dependency Injection** | Injects dependencies into the bean | @Autowired | @Autowired private Order order; |
| **Post-Construction** | Executes tasks after dependencies are injected | @PostConstruct | @PostConstruct public void init() {} |
| **Bean Usage** | Bean is used in application logic | - | userService.performTask(); |
| **Pre-Destruction** | Executes cleanup tasks before bean destruction | @PreDestroy | @PreDestroy public void destroy() {} |

### **Why is the Bean Lifecycle Important?**

1. **Custom Initialization**: Perform custom tasks using @PostConstruct.
2. **Resource Management**: Release resources using @PreDestroy (e.g., closing database connections).
3. **Debugging**: Understanding the lifecycle helps in diagnosing issues during bean creation or destruction.
4. **Efficient Use of Resources**: Lazy initialization (@Lazy) avoids unnecessary resource allocation.

### **Detailed Walkthrough of the Bean Lifecycle**

#### **1. Bean Discovery (@ComponentScan)**

* Spring scans the specified package and its sub-packages to identify classes annotated with @Component, @Service, @Repository, or @Controller to register them as beans.

##### **Example: Configure @ComponentScan**

@Configuration

@ComponentScan(basePackages = "com.example")

public class AppConfig {

// Configuration class

}

* **Purpose**: Helps the IoC container locate classes that should be managed as beans.

#### **2. Bean Creation**

* Spring creates an instance of the bean using the **default constructor** or a custom constructor if defined.

##### **Example**

@Component

public class User {

public User() {

System.out.println("User bean is created");

}

}

**Console Output**:

User bean is created

#### **3. Dependency Injection**

* After creating the bean, Spring resolves and injects the dependencies annotated with @Autowired.

##### **Example with Field Injection**

@Component

public class User {

@Autowired

private Order order;

public User() {

System.out.println("User bean is created");

}

}

* Spring looks for a bean of type Order in the application context and injects it into the User bean.

##### **Example with Constructor Injection**

@Component

public class User {

private final Order order;

@Autowired

public User(Order order) {

this.order = order;

System.out.println("User bean is created with Order dependency");

}

}

#### 

#### 

#### 

#### **4. Post-Construction (@PostConstruct)**

* The method annotated with @PostConstruct is invoked after dependencies are injected and the bean is fully initialized.

##### **Example**

@Component

public class User {

@Autowired

private Order order;

public User() {

System.out.println("User bean is created");

}

@PostConstruct

public void init() {

System.out.println("User bean is fully initialized");

}

}

**Console Output**:

User bean is created

Order bean is created

User bean is fully initialized

#### **5. Bean Usage**

* At this stage, the bean is ready for use. It can participate in business logic and other operations.

##### **Example: Use Bean in a Service**

@Component

public class UserService {

@Autowired

private User user;

public void performTask() {

System.out.println("Performing a task with the User bean");

}

}

#### **6. Pre-Destruction (@PreDestroy)**

* The method annotated with @PreDestroy is invoked during the shutdown of the application context to release resources or perform cleanup.

##### **Example**

@Component

public class User {

@Autowired

private Order order;

public User() {

System.out.println("User bean is created");

}

@PreDestroy

public void destroy() {

System.out.println("User bean is being destroyed");

}

}

**Console Output on Application Shutdown**:

User bean is being destroyed

### 

### 

### **Complete Example with Full Lifecycle**

Here’s a full example that integrates all the lifecycle stages:

#### **1. Order Class (Dependency)**

@Component

public class Order {

public Order() {

System.out.println("Order bean is created");

}

}

#### **2. User Class (Dependent Bean)**

@Component

public class User {

@Autowired

private Order order;

public User() {

System.out.println("User bean is created");

}

@PostConstruct

public void init() {

System.out.println("User bean is fully initialized");

}

@PreDestroy

public void destroy() {

System.out.println("User bean is being destroyed");

}

}

#### **3. Configuration Class**

@Configuration

@ComponentScan(basePackages = "com.example")

public class AppConfig {

}

#### **4. Main Application**

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class MainApp {

public static void main(String[] args) {

// Initialize Spring Context

AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

// Retrieve the User Bean

User user = context.getBean(User.class);

// Close the context to trigger @PreDestroy

context.close();

}

}

#### **Console Output**

Order bean is created

User bean is created

User bean is fully initialized

User bean is being destroyed

### **Key Concepts to Remember**

1. **@Lazy Initialization**:  
   * A bean marked with @Lazy will only be created when it is accessed for the first time.
2. **Singleton Scope (Default)**:  
   * By default, Spring beans are singletons, meaning only one instance is created for the entire application context.
3. **Prototype Scope**:  
   * For prototype beans, a new instance is created every time the bean is requested.

##### **Example: Change Scope**

@Component

@Scope("prototype")

public class Order {

public Order() {

System.out.println("Order bean is created");

}

}

### **Common Mistakes to Avoid**

1. **Forgetting to Close Context**:  
   * If the application context is not closed, @PreDestroy methods won’t be executed.
   * Use context.close() in non-web applications.
2. **Misconfigured Scans**:  
   * Ensure @ComponentScan is configured to scan the correct package for beans.
3. **Dependency Resolution Issues**:  
   * Ensure there is a matching bean in the context for every @Autowired dependency.

### **1. Spring Bean Scopes**

Bean scope defines the lifecycle and visibility of a bean within the application context. Spring provides several scopes for beans:

| **Scope** | **Description** |
| --- | --- |
| **Singleton** | (Default) A single instance of the bean is created and shared across the application. |
| **Prototype** | A new instance is created every time the bean is requested. |
| **Request** | A single instance is created for each HTTP request (used in web applications). |
| **Session** | A single instance is created for each HTTP session (used in web applications). |
| **Application** | A single instance is created for the entire servlet context (used in web applications). |
| **WebSocket** | A single instance is created for the WebSocket lifecycle (used in web apps). |

#### **Example: Changing Scope**

@Component

@Scope("prototype") // Use prototype scope for creating new instances each time

public class Order {

public Order() {

System.out.println("Order bean is created");

}

}

The @Lazy annotation delays bean creation until it is explicitly required.

#### **Example: Lazy Initialization with Dependency**

@Component

public class User {

@Autowired

@Lazy // Dependency is created only when accessed

private Order order;

public User() {

System.out.println("User bean is created");

}

public void accessOrder() {

System.out.println("Accessing Order bean");

System.out.println(order);

}

}

#### **Main Application**

public static void main(String[] args) {

AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

User user = context.getBean(User.class); // Only User bean is created

user.accessOrder(); // Order bean is created here

context.close();

}

**Output**:

User bean is created

Accessing Order bean

Order bean is created

### **8. Full Lifecycle Flow Summary**

Here’s the full lifecycle flow with all the concepts integrated:

1. **Application Context Initialization**:  
   * @ComponentScan identifies beans.
   * Beans are created (@Component or other annotations).
2. **Dependency Injection**:  
   * Spring injects dependencies (@Autowired, @Qualifier).
3. **Post-Construction**:  
   * @PostConstruct or afterPropertiesSet() is called.
4. **BeanPostProcessor**:  
   * Custom logic is executed before and after initialization.
5. **Bean Usage**:  
   * Beans participate in application logic.
6. **Pre-Destruction**:  
   * @PreDestroy or destroy() is invoked for cleanup.
7. **Prototype Bean Cleanup**:  
   * Manually destroy prototype beans if needed.