Spring :

Principles of Object-Oriented Design - **SOLID Principles:**

1. S: The Single Responsibility Principle (SRP)

Definition: A class should have only one reason to change — it should have one job or responsibility.

Why: Reduces complexity, improves readability, and simplifies testing

1. O: The Open - Closed Principle (OCP)

Definition: Software entities (classes, modules, functions) should be open for extension but closed for modification.

Why: Helps add new functionality without altering existing code, reducing bugs and side effects

1. L: The Liskov Substitution Principle (LSP)

Definition: Subtypes must be substitutable for their base types without altering the correctness of the program.

Why: Ensures inheritance is used correctly — child classes shouldn’t break parent class behavior

If S is a subclass of T, then objects of type T should be replaceable with objects of type S **without breaking** the functionality.

1. I: The Interface Segregation Principle (ISP)

Definition: Clients shouldn’t be forced to depend on methods they don’t use.

Why: Helps keep interfaces lean and focused.

1. D: The Dependency Inversion Principle (DIP)

Definition: **High-level modules** should not depend on **low-level modules**. Both should **depend on abstractions (interfaces).**

Why: Decouples classes, making systems more flexible and easier to maintain.

**What is IOC (Inversion of Control)**

1. IoC (Inversion of Control) is a design principle where the control of object creation and dependency management is transferred from the application to the framework.
2. In Spring Boot, IoC is implemented using the Spring IoC Container.

**The IoC container in Spring is responsible for:**

1. **Creating and managing objects** (beans).
2. **Handling dependencies** between objects.
3. **Configuring and injecting dependencies** automatically

**Types of IoC Containers in Spring**

**BeanFactory**

1. The simplest container that provides basic dependency injection.
2. Uses lazy loading (creates beans only when requested).
3. Example: XmlBeanFactory (deprecated).

**ApplicationContext(Recommended)**

1. More advanced than BeanFactory.
2. Supports eager loading (creates beans at startup).
3. Provides additional features like event propagation, declarative bean creation, and AOP.

**Common implementations:**

* **AnnotationConfigApplicationContext –** for Java-based configuration**.**
* **ClassPathXmlApplicationContext** – for XML-based configuration.
* **WebApplicationContext –** for web applications.

**How IoC Works in Spring Boot / How bean is created**

When a Spring Boot application starts, it **automatically creates an ApplicationContext** and performs **component scanning** in the base package (and its sub-packages).  
It scans for classes annotated with **stereotype annotations**. These classes are detected, registered as beans, and managed by the Spring IoC container.

1. **@Component –** Generic bean declaration.
2. **@Service –** Marks a service layer component.
3. **@Repository –** Marks a repository (DAO) layer component.
4. **@Controller / @RestController –** Marks a web controller.
5. **@Bean –** Declares a bean inside a @Configuration class.

then it creates a bean of classes and store in logical memory in form of key and value, this logical memory called as IOC container. Where Key will be the class name in camel case and value will be the **instance of that class**.

**How can we create a bean ?**

1. Using **Stereotype annotations –** This is the most common way in Spring Boot. Spring will **automatically detect** this class during component scanning. It becomes bean in ApplicationContext.
2. Using @Bean inside a @Configuration class – This is more manual, but gives full control over how the bean is created.
3. Using XML Configuration – Old way, rarely used. Define bean in XML file and in java code

**What is Dependency?**

a **dependency** is an object that another object relies on.

**What is Dependency Injection (DI)?**

Dependency Injection (DI) is a **design pattern** where the dependencies of a class are provided (injected) from the outside rather than the class creating them internally.

**Advantages of DI :**

1. Promotes **loose coupling**
2. Makes code **easier to test** (unit testing)
3. Supports **Inversion of Control (IoC)** principle
4. Encourages **better design practices**

**Type of Dependency Injection**

1. **Field / Autowired injection** -> it is slow bcoz it uses reflection, uses @Autowired directly on fields
2. **Setter injection** -> Injects dependency through setter method. it is faster, object is optional
3. **Constructor Injection (Recommended)** -> Injects dependency through constructor. object is mandatory i.e. @Autowire & @Service @Autowired is optional, ensures immutability and mandatory dependencies.
4. **Look up method Injection**

**To reduce the ambiguity – when you have more than one bean of the** same type, Spring doesn’t know **which one to inject.**

**It will throw “NoUniqueBeanDefinitionException: expected single matching bean but found 2..."**

**So, we use –**

1. use **@Primary** on class
2. Use **@Qualifier(“beanName”)** on object in controller class

It works **along with** **@Autowired**.

1. Name object similar to that of the object of class that we want

**BeanInstantiationException in Spring Boot**

A **BeanInstantiationException** in Spring Boot occurs when the **Spring IoC container** is unable to instantiate a bean. This typically issues related to dependencies or class structure.

Commonly caused by – **Missing Dependencies, no default constructor**

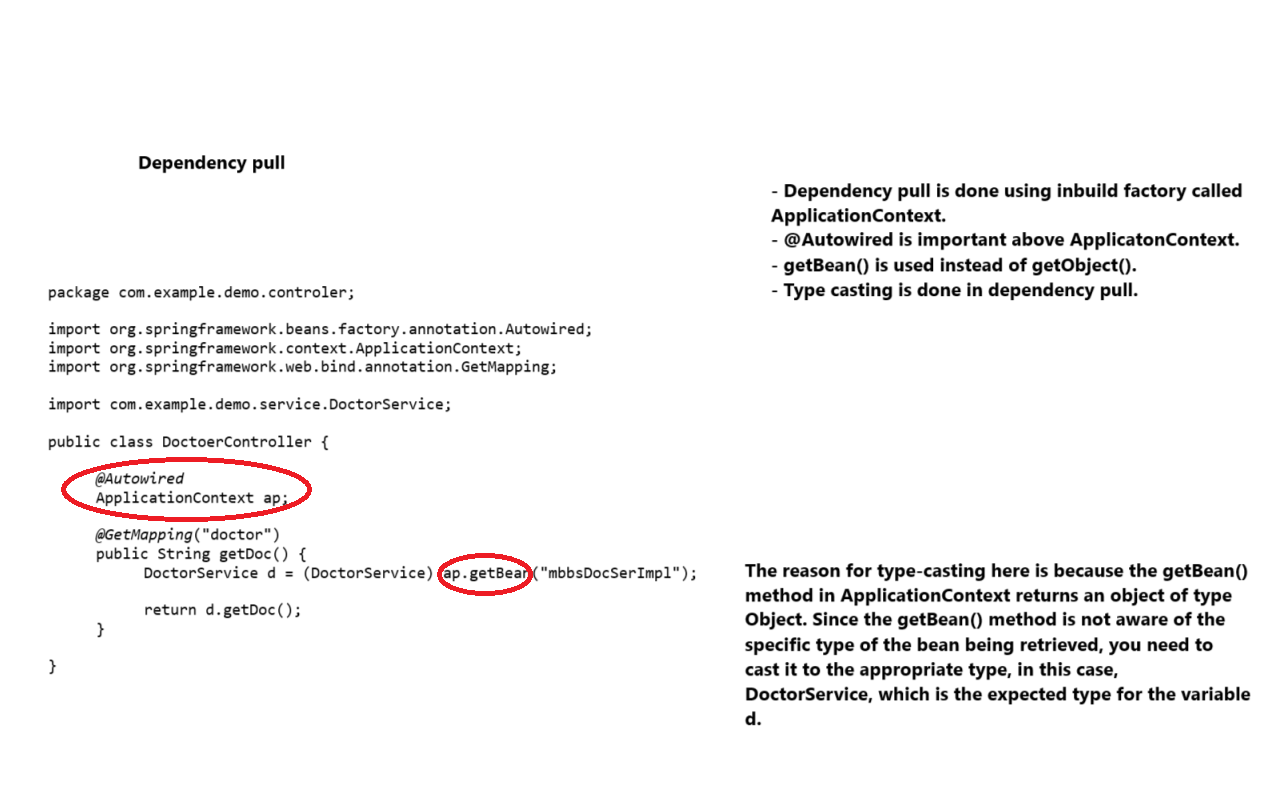
Can be resolved using – Ensure constructor dependencies are present, provide default constructor, Ensure proper bean registration.

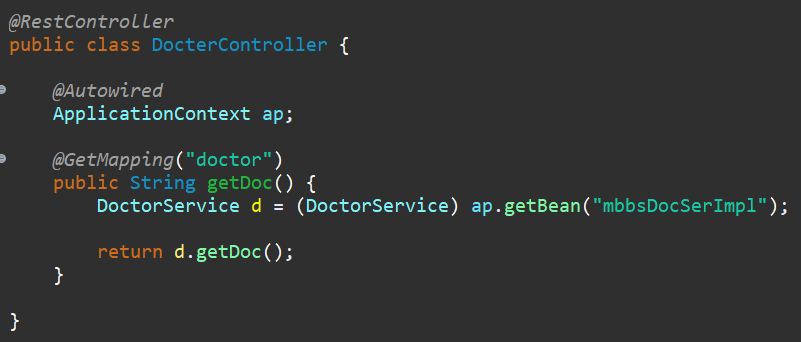
**What is Dependency Pull?**

In Dependency Pull, a class creates or fetches its own dependencies internally.

**Disadvantages of (DP) :**

1. Harder to test (can't mock easily)
2. Tightly coupled
3. Less flexible
4. Violates **Single Responsibility Principle**





**Design Patterns :**

Design patterns are well proved solution of commonly occurring problems in software design.

Design patterns represent an idea.

**Creational Design Pattern** – Factory Pattern, Builder Pattern, Singleton Pattern etc

**Structural Design Pattern** – proxy Pattern, Adaptor Pattern etc

**Behavioral Design Pattern** – Template Pattern, Observer Pattern, Strategy Pattern, Iterator Pattern etc

1. **Singleton design pattern.**

Singleton ensures that **only one instance** of a class is **created during the entire lifecycle** of an application, and provides a **global point of access** to that instance via the getInstance() method.

It typically includes

* A private static **reference of its own type** (to hold the single instance).
* A private constructor (to prevent instantiating from outside).
* A public static method (commonly called getInstance() or getObject(), which provides access to the single instance.

When to use Singleton?

1. When the object is common and sharable or stateless(no data) then we can create that object as Singleton.
2. Only one object at JVM level
3. Every instance of that object will have same hashCode

**Singleton Bean :**

It is created by IOC Container. There will be only single bean at IOC level.



**By default every bean scope is singleton**



In simple cases, the static method checks if the instance exists and creates it if not. In multithreaded environments, **double-checked locking** is used with a **synchronized block** to ensure that only one instance is created, maintaining **thread safety** and **lazy initialization**. This way, the **singleton pattern** guarantees that only one shared object exists throughout the application.

**Lazy Singleton (maybe slow):**

* The instance is created only when needed
* Can be made thread safe using synchronization.
* More complex Must handle thread safety carefully.

**Eager Singleton (Fast):**

* An instance is created at the time of class loading.
* It is simple and thread safe by default
* Instance is created even it is never used causing unnecessary memory utilization.
* **Used in hibernate util to create SessionFactory object and loggers.**



Synchronized Method :

Simple to implement and thread safe.

Disadvantages :

Slower performance because every call to getInstance() is synchronized, even after the instance is created.



Synchronized Block **(Lazy initialization and Double-checked locking )** :

Thread safe.

Better Performance : Synchronization is only used when instance is being created (first time).

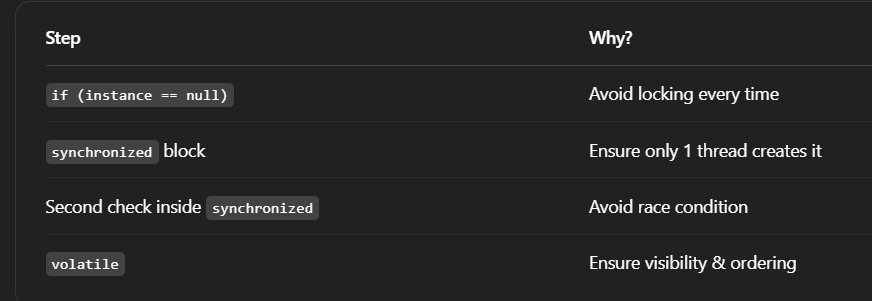
**Uses Double-checked locking to avoid unnecessary synchronization after the instance is initialized**.

**Disadvantages :**

Slightly more complex.

Requires volatile to ensure proper visibility of instance across the threads.

**Double check locking**



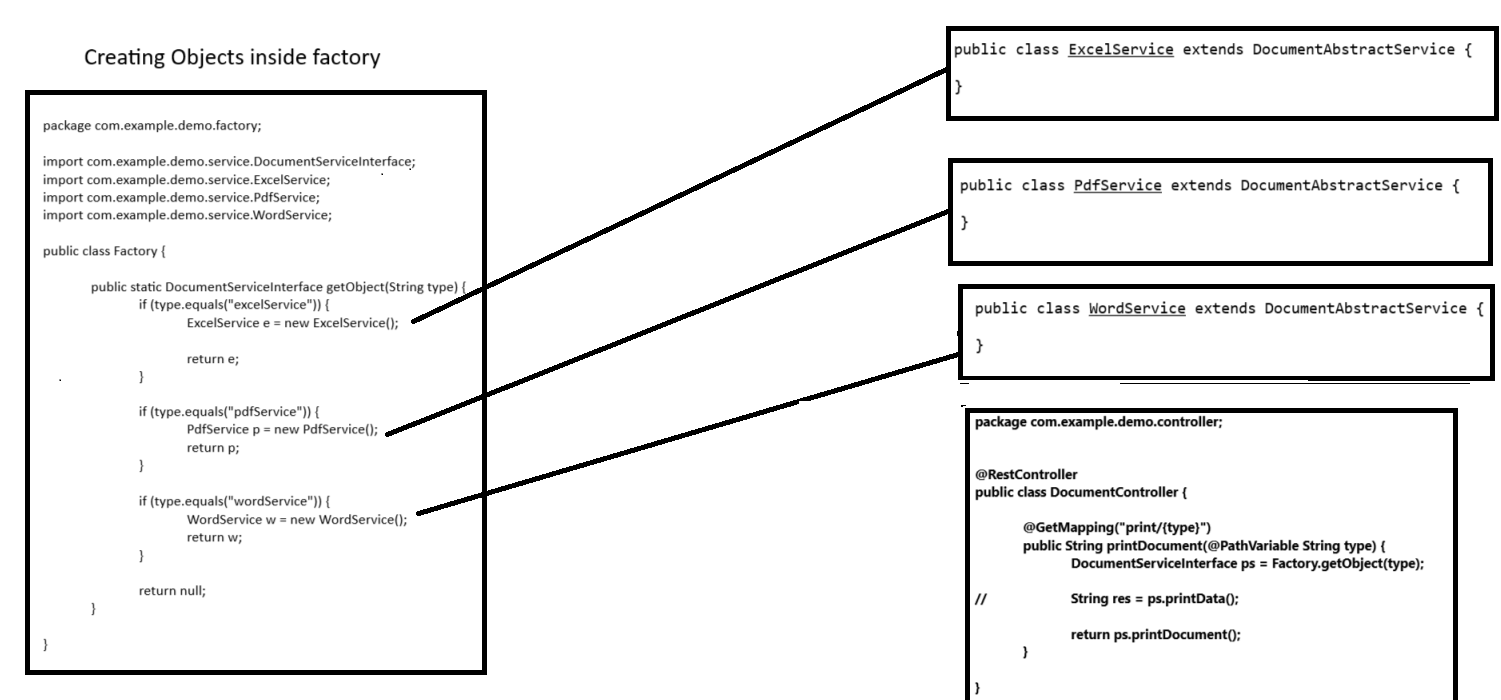
1. **Factory Method Design Pattern (Creational Design Pattern)**

Factory Pattern is used to **create objects** without directly using the new keyword and make our application loosely coupled. It gives us a **central place (a factory class)** to create different types of objects based on input.

**Example:**  
Imagine a notification system:

* If you want to send an **Email**, the factory gives you an EmailNotification object.
* If you want to send an **SMS**, it gives you an SMSNotification object.

You just ask the factory, and it gives the right object.



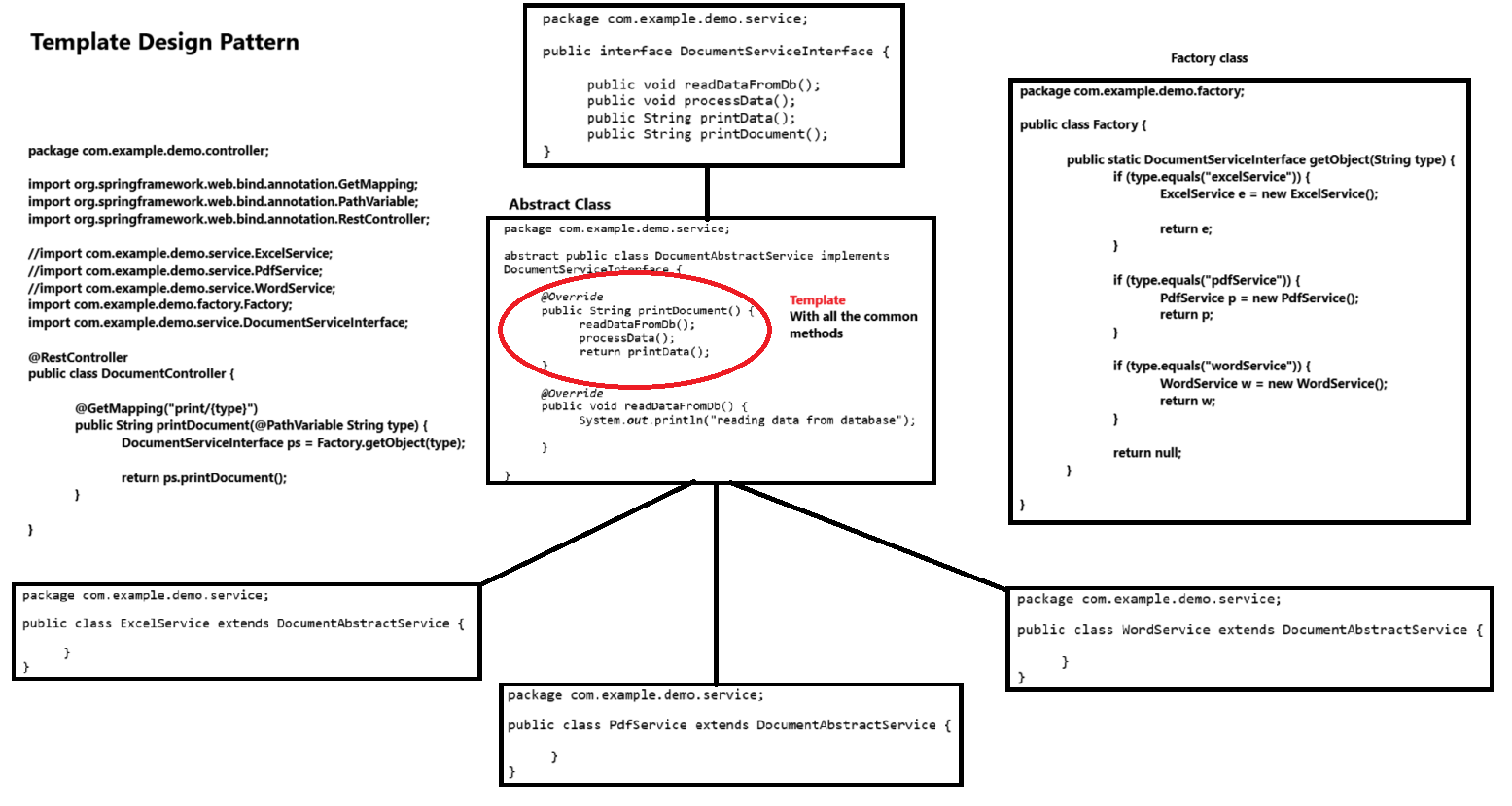
1. **Template Method Design Pattern (Behavioral Design Pattern)**

Template Pattern is used when you have a **common process** but some steps of it can change. You write the **fixed steps in a base class**, and **let child classes fill in the blanks**.

**Example:**  
Imagine you have to **process data**:

* The steps are always: read → process → write.
* For PDF, you read PDF and process PDF.
* For Excel, you read Excel and process Excel.

So the main steps stay the same, but **how** you do each step can vary — that’s where Template Pattern helps.



1. **Builder Design Pattern :**

While creating object when object contain many attributes there are many problems exists:

1. We have to pass many arguments to create object.
2. Some parameters might be optional.
3. Factory class takes all responsibility for creating object. If the object is heavy then all complexity is the part of factory class.

So, in builder pattern we create object step by step and finally return final object with desired values of attributes.

Mainly, helps in creating immutable objects

**How It Works**

* You create a **static nested Builder class**.
* The builder sets values using **method chaining**.
* The final object is created by calling the build() method.

**Difference between @Service and @Component:**

**@Componemt:**

1. Definition: generic bean managed by the spring boot.
2. Purpose: Marks a class as a Spring Bean, allowing it to be autodetected during classpath scanning.
3. Use Case: Can be used for any Spring-managed component.

**@Service:**

1. **Definition**: A **specialized version of @Component**, indicating that the class holds **business logic or service-layer operations**.
2. **Purpose**: Improves **readability and clarity** by marking the class as a **service**.
3. **Use Case**: Used in the **service layer** of an application.

**Difference Between @PostMapping and @GetMapping in Spring Boot**

**@GetMapping**

1. HTTP Method : GET
2. Retrieve data from the server.
3. Reads data from the URL or query parameters.

**@PostMapping:**

1. HTTP Method : POST
2. Send data to the server (create/update resources).
3. Sends data in the **request body**.

**Which pdf generator dependency u used to create pdf and excel generator dependency.**

To generate **PDFs** and **Excel files** in a Spring Boot application, we commonly use:

**PDF** : Apache PDFBox

**Excel**: Apache POI, org.apache.poi:poi-ooxml

**difference between Restcontroller and Controller**

**@RestController:**

1. Used for building RESTful APIs.
2. Returns **JSON/XML** (default).
3. @RestController = @Controller + @ResponseBody.

**@Controller:**

1. Used for handling web pages (MVC).
2. Returns HTML/JSP pages.
3. Only @Controller, needs @ResponseBody for JSON.

**Explain how you will make an api to save department**

add dependencies like Spring Boot Starter, Spring data JPA. Write application properties. Add entity class for Department. Create Department repository, DepartmentService, DepartmentController where we will write saveDept method using @PostMapping

**Where you store Environment specific properties?**

We store environment-specific properties in application.properties or application.yml. For different environments like **dev**, **test**, or **prod**, we use profile-specific files such as application-dev.properties or application-dev.yml. These profiles are activated using the spring.profiles.active property. Spring Boot also allows overriding configuration properties using **environment variables** or **command-line arguments**, which is especially useful for production environments and CI/CD pipelines.

**@Scheduled (Scheduling) :**

**Scheduling**  means **running a task automatically at a specific time** or **repeating a task after a certain period** without manual intervention.

E.g. Sending a daily email at 6 AM automatically.

**How it is done?**

Spring Framework provides **built-in support** for scheduling task using:

**@EnableScheduling** – to turn on scheduling in your project.

**@Scheduled** – to schedule methods to run automatically.

Steps

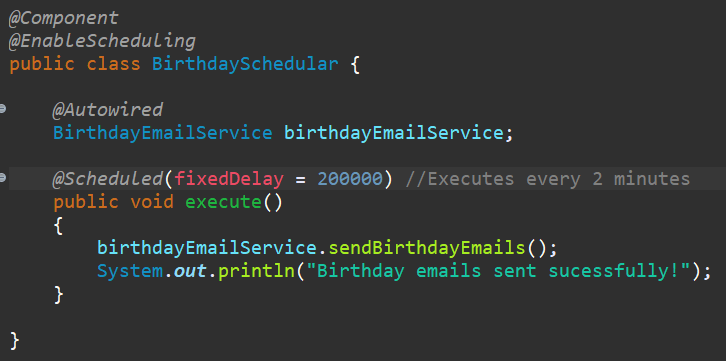
**Step 1**. Enable Scheduling

**Step 2.** Create a Scheduled method.

**@Component –** is used to register the class as a Spring Bean.

**@Scheduled –** tells Spring to run this method automatically based on the timing.

**It is an annotation provided by Spring to run a method at a fixed time interval or based on a corn expression.**

****

**It provides 3 main options :**

**fixedRate →**

* run repeatedly based on start time.
* Runs the task **every X millisecond**, no matter what.
* Start every 5 seconds, even if previous one is still running.
* Overlap happens if task takes longer than the fixedRate interval.

**fixedDelay →**

* run repeatedly after the previous task finishes**.**
* Runs the task **X milliseconds after the last execution finishes.**
* Wait for task to finish, then wait 5 seconds, then start again.
* No overlap happens as next execution starts only after the previous execution + delay time.

**Corn →** run based on complex time expression (like “every Monday at 10 AM”)

**What is Reflection in Java?**

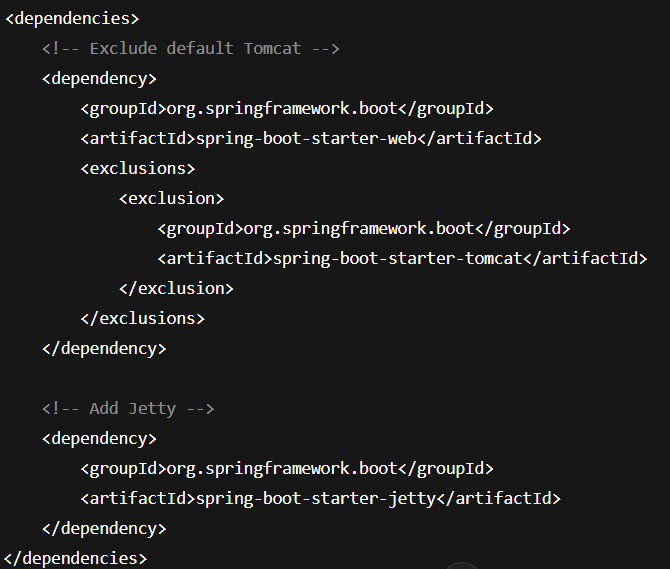
**Reflection** is a feature in Java that allows a program to **inspect** and **manipulate** classes, methods, fields, and constructors **at runtime**, even if they are private.

* Provided by java.land.reflect package.
* We can create new objects, access private fields and methods, invoke methods dynamically, analyze class methods (like annotations, modifiers).
* Used in Frameworks (Spring, Hibernate), Dependency Injection, Serialization/Deserialization, Testing (Junit), Building IDEs and Debuggers.
* It is slower as compared to direct code, breaks encapsulation, can lead to security vulnerabilities if misused.

**Can we change the embedded Tomcat server to another server?**

Yes, we can change the server by adding dependency of the server that we want and disable the tomcat server.

Exclude the previous server and add the dependency of required server.

****

**Spring Boot Features :**

1. **Auto-Configuration**: Automatically configures beans based on classpath and properties, reducing manual setup.
2. **Embedded Servers**: Supports embedded web servers like Tomcat, Jetty, and Undertow.
3. **Minimal Configuration**: Provides sensible defaults, minimizing the need for verbose configuration files.
4. **Starter Projects**: Pre-configured templates for common use cases (e.g., web, data, security).
5. **Production-Ready**: Built-in features like health checks, metrics, and logging for monitoring and managing production apps.
6. **Externalized Configuration**: Supports configuration through properties files, environment variables, or command-line arguments.
7. **Spring Boot Actuator**: Provides monitoring and management endpoints (e.g., /health, /metrics).
8. **DevTools**: Improves development with features like auto-restart and live reload.
9. **Easy Microservices Integration**: Seamlessly integrates with Spring Cloud for building microservices. **Spring Initializer**:

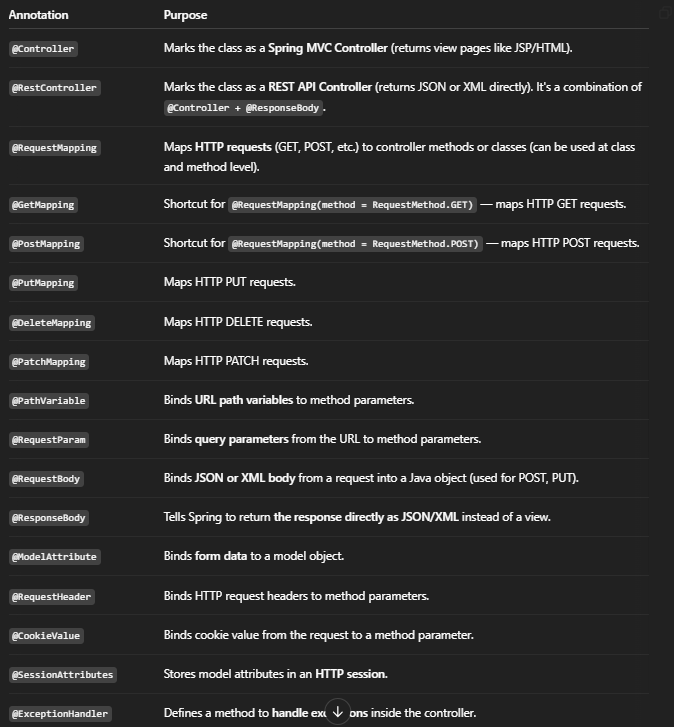
**What is @Configuration vs @AutoConfiguration? How does Spring Boot auto-configuration work?**

| **Feature** | **@Configuration** | **@AutoConfiguration** |
| --- | --- | --- |
| **Purpose** | Manually configures beans in a Spring application. | Automatically configures beans based on application context. |
| **Used In** | **Spring** (both Spring Framework and Spring Boot). | **Spring Boot** (specifically for auto-configuration). |
| **Configuration Style** | Manual configuration using @Bean methods. | Automatic configuration based on classpath and environment. |
| **Annotations Used** | Can be combined with @Bean methods for explicit configuration. | Relies on **@EnableAutoConfiguration** or @SpringBootApplication. |
| **Example** | java @Configuration public class AppConfig { @Bean public MyService myService() { return new MyServiceImpl(); } } | java @SpringBootApplication public class MyApp { public static void main(String[] args) { SpringApplication.run(MyApp.class, args); } } |
| **Flexibility** | You define everything manually. | Spring Boot does the configuration for you automatically. |
| **Customizability** | Full control over bean creation and initialization. | Conditional configuration based on classpath, properties, etc. |

**Spring vs Spring Boot**

|  | **Spring** | **Spring Boot** |
| --- | --- | --- |
|  | A **core framework** for building enterprise-level Java applications | A **rapid application development framework** built on top of Spring |
|  | Manual bean creation and wire using XML or **@Configuration** + **@Bean** in Java classes | Uses component scanning and annotations like **@Component**, **@Service, @Autowired** for automatic wiring |
|  | Developer is responsible for setting up the entire application context, view resolvers, properties | Uses **auto-configuration** to detect beans and configure them based on classpath and application.properties |
|  | Requires an **external servlet container** like Tomcat or Jetty for deployment | Comes with **embedded servers** (Tomcat, Jetty, Undertow), enabling standalone applications |
|  | No built-in tools for monitoring or health checks — must be manually integrated | Includes Spring Boot Actuator, which provides endpoints for health, metrics, logging, etc. |
|  | Large-scale, enterprise-grade applications needing fine control over configuration | Microservices, REST APIs, and cloud-native apps needing faster development with minimal boilerplate |

**Annotations used in controller**

****

**Internal Flow of HTTP Request and Response Handling in a Spring Boot Application (Using Spring MVC)**

When we send the URL like <http://example.com> to the browser, browser first check if it already knows the IP address, if not, it uses DNS to resolve domain name and retrieves IP address. Browser then set up TCP connection with server, if its HTTPS then browser performs TSL/SSL handshake to establish secure connection.

Then browser sends the HTTP request to the server which includes HTTP method (POST, GET), header, and sometimes request body if it’s a POST / PUT request. There is a server called Tomcat server. Server then pass the request to application deployed on it. Inside the application there is a **DispatcherServlet**, that acts as an entry point for all requests, it looks up the request path using **HandlerMapping.** HandlerMapping identifies the appropriate controller and method, request is then passed to the controller to generate the HTTP response.

If request is for webpage, then **ViewResolver** generate the view in HTML page, if request is from postman, then view will be in JSON format.

Once the HTTP response is generated it is send to the web application with **status code, response header (content type) and response body.**

Browser check the content type in the response body and render it accordingly.

**This whole process genarally takes few hundred milliseconds.**

**Bonus point : "**Also, if the server or API uses caching, compression (like GZIP), or Load Balancers, those components can further optimize how fast and efficiently the HTTP response reaches the browser.

**Serialization :** Java object 🡪 byte stream (JSON/XML)

**Serialization** is a process of **converting a Java object to byte stream (or JSON/XML)**

* Easy to **Store** in database, **transferred** between services, or **cached** in memory and **message queues**.
* Java uses the **Serializable interface** for standard serialization.
* In **Spring Boot**, JSON serialization is usually handled by **Jackson Library**.

**Jackson Annotations:**

* @JsonIgnore – skips a field during serialization.
* @JsonProperty – renames a field in JSON.
* @JsonFormat – formats date/time fields.

**Deserialization :**

**Deserialization is the process of converting JSON/XML or byte stream back into a Java object.**

**In Spring Boot:**

* Happens when JSON from a **POST request** is mapped to a Java object.
* Also used in **Kafka/message queues** to convert messages into objects.

**Uses Jackson (by default):**

* @JsonProperty – maps JSON key to Java field.
* @JsonCreator – customizes object creation during deserialization.

**Questions :**

**What is a Singleton class?**

**What are SOLID Principles?**

**What is Dependency Injection (DI)?**

**What is Inversion of Control (IoC)?**

**What is a bean scope and how can it be changed?**

**What is the @Qualifier annotation?**

**What is AOP in Spring Framework?**

**What is the difference between Spring and Spring Boot?**

**What are the key features of Spring Boot?**

**How does Spring Boot support externalised configuration and what are the best practices?**

**Explain the Spring Bean Lifecycle. What are the roles of @PostConstruct and @PreDestroy?**

**How do you secure a Spring Boot REST API using Spring Security?**

**Actuator endpoints and security**

**Spring Dev Tools**

**In Spring Boot, how does auto-configuration work? Can you disable it?**

It uses the **@EnableAutoConfiguration** annotation (often included via @SpringBootApplication).

**Globally** by excluding in @SpringBootApplication:

* @SpringBootApplication(exclude = {DataSourceAutoConfiguration.class}.

**For a specific class** using @EnableAutoConfiguration:

* @EnableAutoConfiguration(exclude = {SecurityAutoConfiguration.class})