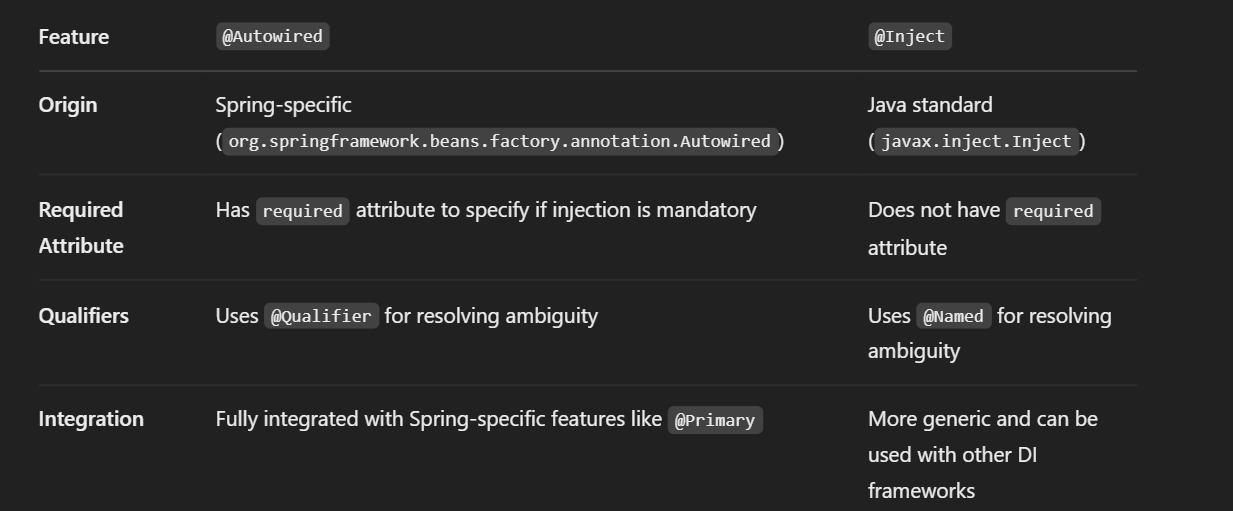
What is the Spring Framework and its core features?

The **Spring Framework** is a comprehensive Java framework that simplifies enterprise application development. Its core features include:

1. **Inversion of Control (IoC) & Dependency Injection (DI):** Manages object creation and dependency injection for loose coupling.
2. **Aspect-Oriented Programming (AOP):** Separates cross-cutting concerns like logging and security from business logic.
3. **Transaction Management:** Simplifies managing database transactions.
4. **Spring MVC:** A framework for building web applications using the Model-View-Controller pattern.

What is the difference between @Autowired and @Inject in Spring?



What is Spring IoC container?

The **Spring IoC (Inversion of Control) container** is a core concept in the Spring Framework. It manages the lifecycle and configuration of objects in a Spring-based application. The container uses **dependency injection (DI)** to supply objects that an application needs, reducing the coupling between components and promoting loose coupling and better testability.

What is a Spring Bean and what are the different scopes of Spring Beans?

In the context of the Spring Framework, a **Spring Bean** is an object that is managed by the **Spring IoC (Inversion of Control) container**. These beans are typically the core components of an application (e.g., services, controllers, DAOs) that are created, configured, and managed by the container.

A Spring Bean can be defined in several ways:

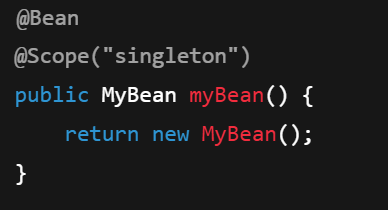
1. **XML Configuration** (traditional way).
2. **Java-based Configuration** using annotations like @Bean and @Configuration.
3. **Component Scanning** using annotations like @Component, @Service, @Repository, and @Controller.

**Different Scopes of Spring Beans:**

Spring allows you to define the **scope** of a bean, which determines the **lifetime** and **visibility** of that bean within the container. The default scope is **singleton**, but Spring supports several other scopes:

**Singleton (Default)**

* **Description:** There is only **one instance** of the bean in the Spring IoC container. This instance is shared across the entire Spring container and reused whenever the bean is requested.



**Prototype**

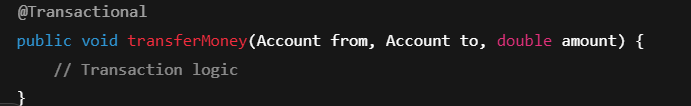
* **Description:** A new **instance** of the bean is created every time it is requested from the container. The container does not manage its lifecycle beyond creation (e.g., it does not handle initialization or destruction).

**Session**

* **Description:** A new **instance** of the bean is created for each HTTP **session**. Similar to the request scope, but the bean's lifecycle is tied to the HTTP session.

In Spring, **transaction management** ensures data consistency and integrity. It supports two approaches:

1. **Declarative Transaction Management**:
   * Most commonly used with the @Transactional annotation.
   * Automatically manages transaction boundaries (commit, rollback) for methods.
   * Can control properties like **propagation**, **isolation level**, and **rollback rules** using the @Transactional attributes.



**Programmatic Transaction Management**:

* Manages transactions manually using TransactionTemplate or PlatformTransactionManager.
* Provides more control but is less common.

A computer screen shot of a black screen

AI-generated content may be incorrect.

Explain the different types of Spring Beans and their lifecycle.

In Spring, a **bean** is any object that is managed by the Spring IoC (Inversion of Control) container. Beans are the core components of an application and can be defined in various ways (XML, annotations, or Java configuration). There are several types of Spring Beans, based on how they are configured, and each type follows a specific **lifecycle**.

**Types of Spring Beans:**

1. **Singleton Bean (Default)**
   * **Description:** There is only one **instance** of the bean per Spring IoC container. This instance is shared across the entire application.
   * **Scope:** singleton (default scope).
   * **Use case:** Ideal for stateless beans where a single instance suffices.
   * **Lifecycle:**
     + The bean is created once when the container is initialized.
     + It is initialized only once and used multiple times throughout the application.
2. **Prototype Bean**
   * **Description:** A new **instance** of the bean is created every time it is requested from the container.
   * **Scope:** prototype.
   * **Use case:** Suitable for stateful beans or when each invocation requires a new instance.
   * **Lifecycle:**
     + A new instance is created each time the bean is requested.
     + Spring does not manage the complete lifecycle (like initialization or destruction) for prototype beans.
3. **Request Bean**
   * **Description:** A new **instance** of the bean is created for each HTTP request in a web application.
   * **Scope:** request (only available in web applications).
   * **Use case:** Typically used in web applications to manage beans specific to a single HTTP request.
   * **Lifecycle:**
     + A new bean is created for each HTTP request.
     + The bean is destroyed when the request ends.
4. **Session Bean**
   * **Description:** A new **instance** of the bean is created for each HTTP session.
   * **Scope:** session (only available in web applications).
   * **Use case:** Used for user-specific data that persists throughout the session.
   * **Lifecycle:**
     + A new bean is created for each HTTP session.
     + The bean is destroyed when the session ends.
5. **Application Bean**
   * **Description:** A single **instance** of the bean is created for the entire lifecycle of the Spring web application.
   * **Scope:** application (only available in web applications).
   * **Use case:** For beans that should be shared across the whole web application context.
   * **Lifecycle:**
     + A single instance is created for the entire application context.
     + It lives until the application is destroyed.

**What are Spring Profiles?**

Spring Profiles are a way to provide a mechanism for conditional configuration in Spring applications. They allow you to define different beans or configurations that should be active based on the environment or use case (e.g., development, testing, production). Using profiles, you can customize the Spring configuration by including/excluding beans depending on the active profile.

A screen shot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer code

AI-generated content may be incorrect.

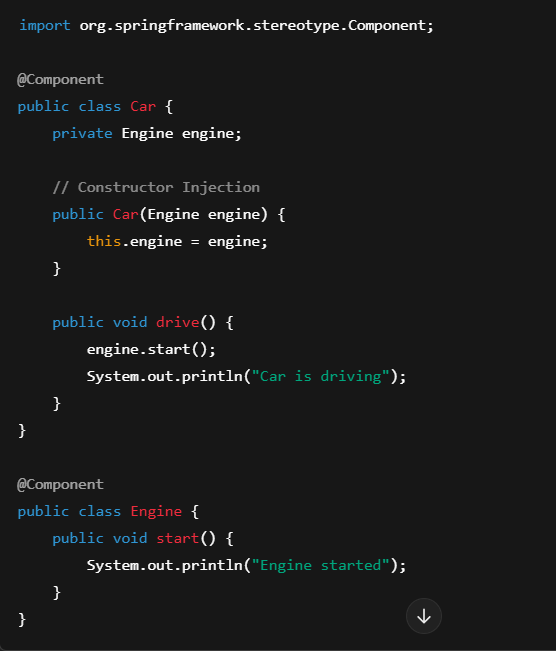
**Dependency Injection (DI)** is a core concept in Spring that allows you to inject dependencies into Spring beans rather than creating them manually within the bean. Spring supports three main types of dependency injection: **Constructor Injection**, **Setter Injection**, and **Field Injection**. Each of these has its use cases and advantages.

**1. Constructor Injection**

In Constructor Injection, dependencies are provided to a bean through its constructor.

**How it works:**

* Spring calls the constructor of the bean and passes the required dependencies as arguments.
* This method is preferred when dependencies are **mandatory** and must be provided at the time of bean creation.



**2. Setter Injection**

In Setter Injection, dependencies are provided through setter methods after the bean is constructed.

**How it works:**

* Spring creates the bean first and then calls setter methods to inject the dependencies.
* This is useful for **optional dependencies** or when you need to modify the dependencies after bean creation.

A screen shot of a computer program

AI-generated content may be incorrect.

**3. Field Injection**

Field Injection allows Spring to inject dependencies directly into fields, without requiring setter or constructor methods.

**How it works:**

* Spring uses **reflection** to directly inject dependencies into the fields.
* Dependencies are marked with @Autowired (or @Inject), and Spring automatically injects the required dependencies.

A screen shot of a computer program

AI-generated content may be incorrect.

A black and white screen with white text

AI-generated content may be incorrect.

**What is Circular Dependency in Spring and how to resolve it (@Lazy)?**

**Circular dependency** occurs when two or more beans depend on each other, either directly or indirectly, forming a cycle. This means that **Bean A** depends on **Bean B**, and **Bean B** depends on **Bean A** (or a chain of dependencies that form a loop). Spring cannot resolve this situation using standard dependency injection because it results in a never-ending cycle.

A screen shot of a computer program

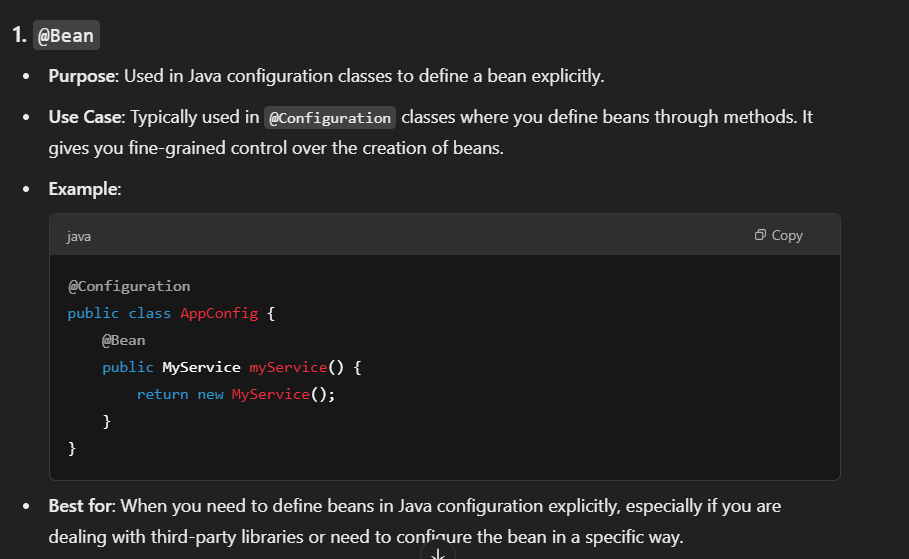
AI-generated content may be incorrect.

**How to Resolve Circular Dependency?**

Spring provides several ways to resolve circular dependencies. One of the most common approaches is using the **@Lazy** annotation.

A screen shot of a computer program

AI-generated content may be incorrect.



A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A black screen with white text

AI-generated content may be incorrect.

A black and white text on a black background

AI-generated content may be incorrect.

A black background with white text

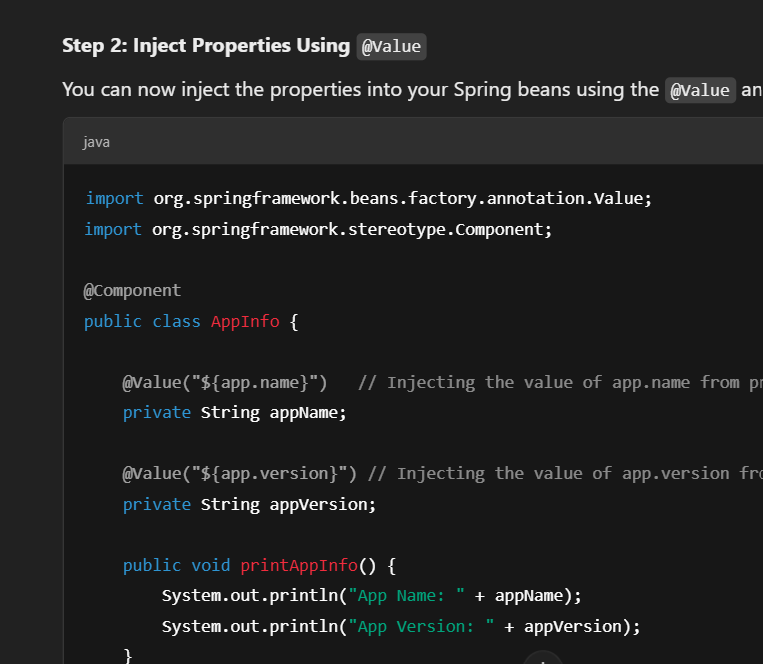
AI-generated content may be incorrect.

app.name=SpringApplication

app.version=1.0.0

A screen shot of a computer

AI-generated content may be incorrect.



**Lombok Annotations: @Data, @NoArgsConstructor, and @AllArgsConstructor**

Lombok is a popular Java library that helps reduce boilerplate code by providing annotations that automatically generate common methods like getters, setters, constructors, toString(), equals(), and hashCode().

**1. @Data**

* **Purpose**: This annotation is a shortcut for several Lombok annotations that automatically generate common methods for a Java class.
* **Generated Methods**:
  + **Getters** for all fields.
  + **Setters** for all non-final fields.
  + **toString()** method.
  + **equals()** and **hashCode()** methods based on all non-transient fields.
  + **A constructor** for required fields (if applicable).

import lombok.Data;

@Data

public class Person {

private String name;

private int age;

}

**2. @NoArgsConstructor**

* **Purpose**: This annotation generates a **no-argument constructor** for the class. It is useful when you need a default constructor that doesn't require any parameters.
* **Use Case**: Often used for **JPA** entities, **Spring Beans**, and when you need a default constructor for frameworks that rely on reflection to instantiate objects.

**3. @AllArgsConstructor**

* **Purpose**: This annotation generates a **constructor with arguments** for all fields in the class.
* **Use Case**: Useful when you want to initialize all fields of a class during object creation, often used in classes that are immutable or require all fields to be set upon instantiation.

**Spring Boot Actuator** is a set of built-in tools and features that provide production-ready functionalities to your Spring Boot application. It helps in monitoring and managing your application in production by exposing a set of endpoints that can provide valuable insights into your application's health, metrics, environment, and more.

Spring Boot Actuator exposes several endpoints out-of-the-box that you can use to monitor and manage the application. Some of these endpoints are enabled by default, while others need to be explicitly enabled or customized in the configuration.

/actuator/health

/actuator/metrics

/actuator/info

/actuator/env