**1. What is ApplicationContext?**

* ApplicationContext is the **central interface** to the **Spring IoC (Inversion of Control) container**.
* It is responsible for **instantiating, configuring, and managing beans**.

**2. What is AnnotationConfigApplicationContext?**

* A specific implementation of ApplicationContext.
* It is used **when you're configuring your Spring application using Java classes**, not XML.
* It **scans and registers** beans based on annotations like @Component, @Configuration, @Bean, etc.

**3. What is AppConfig.class?**

* AppConfig is a class annotated with @Configuration.
* It defines one or more @Bean methods or uses @ComponentScan to discover beans.

**4. Dependency Injection** is a design pattern where **an object’s dependencies (other objects it needs to work)** are **provided from outside**, rather than the object creating them itself.

**🔧 Example Without DI (Tightly Coupled):**

class Car {

Engine engine = new Engine(); // Car creates its own Engine

}

* ❌ Car is tightly coupled to Engine
* ❌ Hard to replace Engine with a different one
* ❌ Harder to test (you can't inject a mock)

**✅ Example With DI (Loosely Coupled):**

class Car {

private Engine engine;

// Dependency is injected via constructor

public Car(Engine engine) {

this.engine = engine;

}

}

Now you can do:

Engine engine = new Engine();

Car car = new Car(engine); // Injecting the dependency

**🔄 Types of Dependency Injection:**

| **Type** | **Description** |
| --- | --- |
| Constructor Injection | Dependencies are provided via constructor |
| Setter Injection | Dependencies are provided via setters |
| Field Injection | Spring injects directly into fields using @Autowired |

5. what does @Configuration stand for

The @Configuration annotation in Spring is used to indicate that a **class contains Spring bean definitions**, i.e., it's a **configuration class**.

**📌 Definition:**

@Configuration

public class AppConfig {

// Bean definitions go here

}

* @Configuration tells Spring that this class is a **Java-based configuration class**.
* It replaces traditional **XML-based configuration** (like applicationContext.xml).

**🧠 What it does:**

* Spring processes the class and looks for methods annotated with @Bean.
* Those @Bean methods define objects that will be **managed as Spring beans**.

**✅ Example:**

@Configuration

public class AppConfig {

@Bean

public Engine engine() {

return new Engine();

}

@Bean

public Car car() {

return new Car(engine()); // injects the Engine bean

}

}

**🔍 What happens here:**

* Spring sees @Configuration, so it knows to look inside AppConfig for bean definitions.
* @Bean methods are called by Spring and the returned objects are **registered in the Spring container**.
* These beans can now be injected anywhere in your app using @Autowired.

**⚠️ Behind the scenes:**

Spring **proxies** the @Configuration class to ensure **singleton beans** are managed correctly. That’s why calling engine() inside car() still returns the **same shared Engine bean**, not a new one each time.

**🆚 Comparison:**

| **Annotation** | **Purpose** |
| --- | --- |
| @Configuration | Marks the class as a source of bean definitions |
| @Component | Marks the class as a Spring-managed bean itself |
| @Bean | Marks a method that returns a bean to be managed by Spring |
|  |  |

6. **Purpose of using @Component annotation in Spring**

The primary **purpose** of the @Component annotation is to:

**Mark a class as a Spring-managed bean** so that Spring can automatically detect it during component scanning and manage its lifecycle (creation, dependency injection, etc.).

**✅ Key Purposes Explained**

1. **Automatic Bean Detection**
   * It allows Spring to **automatically register the class as a bean** without needing manual configuration in XML or Java config.
   * During startup, Spring scans the classpath for classes annotated with @Component.
2. **Dependency Injection**
   * Once registered as a bean, the class can be **injected** into other classes using @Autowired, constructor injection, etc.
3. **Loose Coupling**
   * Helps achieve **loose coupling** between components by using interfaces and injecting implementations managed by Spring.
4. **Simplifies Configuration**
   * No need to define beans explicitly using @Bean or XML <bean> tags.
   * Makes the codebase **cleaner and more maintainable**.

**How to Find Which Classes Should Be Annotated with @Component**

To decide **which classes should be annotated with @Component** (or its specialized forms like @Service, @Repository, @Controller), ask yourself:

**🔍 1. Is this class a candidate for dependency injection?**

If a class is intended to be:

* **Used in other classes**, and
* **Managed by Spring** (you don't want to create its object manually with new),

✅ Then it should be annotated with @Component.

**🧩 2. Does it represent a specific application layer?**

Use more specific stereotypes for better clarity and behavior:

| **Annotation** | **Use for classes that...** |
| --- | --- |
| @Component | Are general-purpose and don’t fall under other types |
| @Service | Contain **business logic** |
| @Repository | Handle **data access** (DAOs, database interactions) |
| @Controller | Handle **web requests** in Spring MVC |

**🔧 3. Is it a custom utility or helper class?**

If the class contains methods or utilities you want to **inject into other beans**, annotate it with @Component.

**📦 4. Is it part of the scanned packages?**

Make sure the class is in a package that is being **scanned by Spring**, either via:

@ComponentScan("com.example.package")

or being under the base package of the main Spring Boot application class.

**Why and Where to Use @Autowired in Spring**

**🎯 Why Use @Autowired**

@Autowired is used to **automatically inject dependencies** in Spring. It tells Spring to:

**Look for a suitable bean** of the required type and **inject it** where needed.

So, instead of creating objects manually using new, Spring takes care of it for you.

**🔧 Where to Use @Autowired**

You can use @Autowired in 3 main places:

| **Place** | **Example** | **Use Case** |
| --- | --- | --- |
| **1. Field injection** | @Autowired private Engine engine; | Simple, but not recommended for large apps (hard to test) |
| **2. Constructor injection** ✅ *Best Practice* | public Car(Engine engine) { this.engine = engine; } | Preferred for immutability and testability |
| **3. Setter injection** | public void setEngine(Engine engine) | Useful when the dependency is optional or can change |

**Why Use @Qualifier**

Spring’s @Autowired injects beans **by type**. But when **multiple beans of the same type exist**, Spring doesn’t know which one to inject.

@Qualifier is used to **specify exactly which bean to inject** when there are multiple candidates of the same type.

**❌ Problem Without @Qualifier**

java

CopyEdit

@Component

public class PetrolEngine implements Engine {}

@Component

public class DieselEngine implements Engine {}

@Component

public class Car {

@Autowired

private Engine engine; // ❌ Spring will throw an error: multiple candidates

}

➡ Spring finds **two beans of type Engine** and gets confused.

**✅ Solution With @Qualifier**

java

CopyEdit

@Component

public class Car {

@Autowired

@Qualifier("petrolEngine") // ✅ Specify which one to inject

private Engine engine;

}

**🧠 How @Qualifier Works**

* @Qualifier works **with @Autowired** to **pinpoint the correct bean**.
* The value inside @Qualifier("...") must match the **bean name** (usually the class name in camelCase).

**📍 When to Use @Qualifier**

Use @Qualifier when:

| **Situation** | **Reason** |
| --- | --- |
| You have **multiple beans of the same type** | To tell Spring **which one to use** |
| You want to **inject a specific implementation** | Especially in cases of strategy patterns or plug-ins |
| You want **fine control over bean injection** | For clarity, even if only one bean exists |

**🔧 Also Works With Constructor Injection**

java

CopyEdit

@Component

public class Car {

private final Engine engine;

@Autowired

public Car(@Qualifier("dieselEngine") Engine engine) {

this.engine = engine;

}

}

**📌 Summary**

| **Annotation** | **Purpose** |
| --- | --- |
| @Autowired | Injects a bean by **type** |
| @Qualifier | Specifies **which bean** to inject when multiple exist |

**SECTION -5**

**The NoUniqueBeanDefinitionException is an exception in Spring Framework that occurs when Spring's dependency injection system finds multiple beans of the same type in the application context, but it doesn't know which one to inject.**

**SECTION-8**

**What is Spring Framework?**

**The Spring Framework is a comprehensive Java-based framework used for building enterprise-level applications. It provides infrastructure support for developing Java applications and promotes good practices such as loose coupling through Dependency Injection (DI).**

**✅ Core Features of Spring Framework:**

* **Dependency Injection (DI): Helps in building loosely coupled components.**
* **Aspect-Oriented Programming (AOP): Separates cross-cutting concerns like logging, security, etc.**
* **Spring MVC: Framework to build web applications using Model-View-Controller architecture.**
* **Transaction Management: Simplifies database transaction handling.**
* **Integration with JDBC, JPA, JMS, etc.: Easy integration with databases and messaging systems.**

**🔹 What is Spring Boot?**

**Spring Boot is a project built on top of the Spring Framework that simplifies the process of creating standalone, production-ready Spring applications. It auto-configures your application and removes the need for a lot of boilerplate configuration.**

**✅ Key Features of Spring Boot:**

* **Auto-configuration: Automatically configures beans based on classpath contents.**
* **Standalone applications: You can run your app with a simple main() method (no need for external servers).**
* **Embedded Servers: Comes with Tomcat, Jetty, or Undertow embedded—no need to deploy WARs.**
* **Spring Boot Starter Dependencies: Pre-packaged dependencies for common tasks (like web, data, security, etc.).**
* **Spring Boot Actuator: Built-in endpoints for monitoring and managing the app.**

**🟠 What is the problem with Spring Framework?**

**Spring Framework is powerful, but it has some problems:**

1. **Too Much Setup:  
   You have to write a lot of code just to start a project.**
2. **Complicated Configuration:  
   You need to configure everything manually — like database, server, etc.**
3. **Needs External Server:  
   You have to deploy your project on an external server like Tomcat.**
4. **Takes More Time:  
   Because of all the setup and configuration, it takes longer to build even a simple app.**

**🟢 How does Spring Boot fix this?**

**Spring Boot makes everything faster and easier:**

1. **Auto Setup:  
   Spring Boot configures most things for you automatically.**
2. **Embedded Server:  
   It has a built-in server (like Tomcat), so you don’t need to install one.**
3. **Fewer Files to Write:  
   You don’t need XML files or long config files — just a few annotations.**
4. **Start Quickly:  
   You can build and run your project with just one file (main() method).**

**🌟 Spring Boot Features (Simplified)**

1. ✅ **Auto Configuration**
   * It sets up things for you automatically (like database, server).
2. 🚀 **Built-in Server**
   * No need to install Tomcat — just run your app directly.
3. 📦 **Starter Packages**
   * Ready-made sets of tools (like web, database, etc.).
4. 📄 **No XML Needed**
   * Uses simple Java annotations instead of long XML files.
5. 📊 **Monitoring with Actuator**
   * Check app health, metrics, etc. easily.
6. 🧪 **Easy Testing**
   * Built-in tools for testing your code.
7. 🌍 **Multiple Environments**
   * You can run different settings for dev, test, and prod.
8. ⚡ **Quick Project Setup**
   * Use [start.spring.io](https://start.spring.io) to create a project fast.

The terms **URI** and **URL** are closely related but not identical. Here's a clear breakdown of the **difference between URI and URL**:

**🔹 URI (Uniform Resource Identifier)**

* A **URI** is a **general identifier** for a resource.
* It can refer to a resource **by name, location, or both**.
* It includes both **URLs** and **URNs** (Uniform Resource Names).

**✅ Examples of URIs:**

* https://example.com/index.html (a URL — because it locates a resource)
* urn:isbn:0451450523 (a URN — names a resource but doesn’t give its location)

**🔹 URL (Uniform Resource Locator)**

* A **URL** is a **specific type of URI**.
* It **locates** a resource by describing its **access method** (protocol) and its **address**.

**✅ A URL always includes:**

1. **Scheme/Protocol** — like http, https, ftp
2. **Location** — like domain or IP
3. Optionally, a **path**, **query**, **port**, etc.

**✅ Example of a URL:**

* <https://www.google.com/search?q=chatgpt>

In Spring, we **extend JpaRepository** to get a **ready-to-use set of methods** for interacting with a database, without having to write SQL or boilerplate code.

**🔹 What is JpaRepository?**

JpaRepository is a part of **Spring Data JPA**, which is a framework that simplifies the implementation of data access layers using **JPA (Java Persistence API)**.

It extends two other interfaces:

JpaRepository<T, ID> extends PagingAndSortingRepository<T, ID> extends CrudRepository<T, ID>

**🔹 Why do we extend JpaRepository?**

Here are the main reasons:

**✅ 1. Built-in CRUD operations**

You get standard methods like:

save(), findById(), findAll(), deleteById(), existsById(), count()

**✅ 2. Custom query generation**

You can define methods like:

List<User> findByEmail(String email);

List<Product> findByPriceGreaterThan(double price);

Spring automatically generates the query for you based on the method name.

**✅ 3. Pagination and sorting support**

You can easily paginate results using:

Page<User> findAll(Pageable pageable);

List<User> findAll(Sort sort);

**✅ 4. Integration with Spring Boot**

Repositories that extend JpaRepository are automatically detected by Spring Boot when you annotate your class with @Repository, and Spring will inject them where needed.

**✅ 5. Custom queries using @Query**

You can write JPQL or native SQL queries like this:

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

List<User> findByName(String name);

**🔸 Example**

public interface UserRepository extends JpaRepository<User, Long> {

List<User> findByEmail(String email);

}

You can now use userRepository.findByEmail("test@example.com") without writing any SQL.

**🔹 Summary**

You extend JpaRepository to:

* Save time and avoid boilerplate
* Get powerful built-in methods
* Support custom queries and pagination
* Integrate easily with Spring Boot and JPA