IOC (Inversion of control)

IoC stands for Inversion of Control. It is a fundamental design principle implemented in the Spring Framework via the **IoC Container** (also known as the Spring ApplicationContext).

In simple terms, IoC means transferring the responsibility of creating and managing objects and their dependencies from **your code** to a **framework** (the Spring Container).

What IoC Does

The primary purpose of IoC is to achieve **loose coupling** between components, making the application more modular, maintainable, and testable.

Feature	Description	Benefit
Inverts Control	In traditional programming, an object is responsible for creating or looking up the other objects it needs (its dependencies). IoC **flips this**; the container creates and provides the dependencies to the object.	You focus only on the business logic; the framework handles the "plumbing."
Dependency Injection (DI)	DI is the concrete technique Spring uses to implement IoC. The container "injects" the dependencies (beans) into the consuming object, typically via a constructor, setter method, or field.	Objects are independent and easily swappable, which is crucial for unit testing (you can easily mock the dependencies).
Lifecycle Management	The container manages the entire lifespan of the objects (beans), from instantiation to configuration, use, and eventual destruction.	Prevents resource leaks and ensures resources are managed correctly across the application.

Export to Sheets

How the IoC Container Manages Beans

The Spring IoC Container acts as a central registry and factory for all the objects (beans) in your application.

1. Reading Configuration Metadata

When the Spring application starts, the IoC Container begins by reading its **configuration metadata**. This metadata tells it which classes to manage and how to connect them. This is typically provided through:

- **Annotations:** (`@Component`, `@Service`, `@Repository`, `@Controller`, `@Configuration`, `@Bean`)
- **Java Code:** (Methods annotated with `@Bean` inside a `@Configuration` class)
- **XML:** (Historically common, now less frequent)

2. Instantiation (Creating the Beans)

Based on the configuration, the container begins to create the bean instances. By default, most beans are created as **Singletons** (only one instance exists for the entire application).

3. Assembling (Dependency Injection)

This is where the "control" is inverted. For every bean it creates, the container looks for any required dependencies (using annotations like `@Autowired`).

- If a `UserService` needs a `UserRepository`, the container first finds the `UserRepository` bean.
- It then "injects" that `UserRepository` instance into the `UserService` instance (often via the constructor).
- The `UserService` bean is now fully assembled and ready for use.

4. Lifecycle Callbacks

The container controls when a bean is initialized and destroyed:

- **Initialization:** After the bean is created and all dependencies are injected, the container calls any initialization methods (e.g., those annotated with `@PostConstruct`) to perform setup tasks.
- **Use:** The bean is ready to be retrieved and used by the rest of the application.

- **Destruction:** When the container shuts down, it calls any destruction methods (e.g., those annotated with `@PreDestroy`) to allow the bean to release resources (like closing database connections).

-

Spring Boot 3.4.2 is a patch release built upon the major Spring Boot 3 line. Its fundamentals are centered on simplifying the creation of production-ready, standalone Spring applications, while leveraging the latest features of modern Java.

The core philosophy is based on **opinionated configuration** to minimize boilerplate code.

1. Core Fundamentals (The "Why" and "How")

The foundation of Spring Boot rests on three key pillars inherited from the Spring Framework, but heavily automated:

A. Inversion of Control (IoC) Container

The IoC Container (or Application Context) is the heart of every Spring application.

- **What it does:** It manages the entire **lifecycle** of Java objects (called **Beans**), including their creation, configuration, and destruction.
- **How it's used:** You define a class with an annotation like `@Component`, `@Service`, or `@Repository`, and Spring takes over the management, eliminating manual instantiation (`new MyClass()`).

B. Dependency Injection (DI)

DI is the mechanism used to implement IoC, ensuring components are loosely coupled.

- **What it does:** Instead of an object creating its own dependencies, the IoC Container automatically "injects" them where they are needed (e.g., using `@Autowired` or constructor injection).
- **Benefit:** This promotes modular, flexible, and highly testable code.

C. Auto-Configuration

This is Spring Boot's most defining feature, automating much of the setup process.

- **What it does:** Based on the libraries you include in your project's classpath (via **Starter Dependencies**), Spring Boot automatically configures the necessary beans.
- **Example:** If you add the `spring-boot-starter-web` dependency, Spring Boot automatically configures an **embedded Tomcat server**, Spring MVC, and the necessary JSON converters—all without you writing any configuration code.

2. Key Features of Spring Boot 3.x

Spring Boot 3.x, and specifically version 3.4.2, introduced major shifts and integrations crucial for modern applications:

Feature	Description	Requirement/Benefit
Java 17+ Baseline	Spring Boot 3.x requires a minimum of **Java 17** (the current LTS) and is built on **Spring Framework 6.x**.	Ensures support for modern Java language features (records, text blocks, pattern matching).
Jakarta EE Migration	It migrated from the older Java EE namespace (`javax.*`) to the **Jakarta EE 10** namespace (`jakarta.*`).	Necessary for compliance with modern Java Enterprise specifications.
Native Image Support	Full support for compiling Spring Boot applications into **GraalVM Native Images** (using Ahead-Of-Time or AOT compilation).	Drastically improves **startup time** and reduces memory footprint, which is critical for microservices and serverless environments.
Starters and Starters	**"Starter" dependencies** are a set of convenient dependency descriptors (e.g., `spring-boot-starter-data-jpa`) that automatically pull in related dependencies with compatible versions.	Simplifies build configuration and avoids version conflicts.
Actuator	Provides production-ready monitoring and management endpoints (like `/health`, `/metrics`, `/env`) out of the box.	Essential for monitoring, health checking, and troubleshooting applications deployed to production.
Externalized Configuration	Allows configuration properties to be externalized (e.g., in `application.properties`, YAML files, environment variables, or command-line arguments).	Enables a single application binary to be deployed across different environments (development, testing, production) with environment-specific settings.

does the @components class beans are auto configured

That's an insightful question about Spring's terminology!

Spring boot auto-configuration mechanism:

The Spring Boot auto-configuration mechanism is the core feature that automatically configures your Spring application based on the dependencies you include in your project. It dramatically reduces the need for manual XML or Java configuration.

What it is and How it Works

Auto-configuration is Spring Boot's way of making **opinionated decisions** for you. It examines the Java libraries and classes present on your classpath and intelligently sets up the necessary Spring beans and configurations.

The mechanism primarily relies on the **`@EnableAutoConfiguration`** annotation (which is part of the convenience annotation **`@SpringBootApplication`**).

1. The Core Steps

1. **Component Scanning:** When the application starts, Spring scans the classpath for auto-configuration classes. These classes are typically located inside the Spring Boot JARs and are listed in a file named `META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports`.
2. **Conditional Logic:** Each auto-configuration class contains beans defined within methods annotated with **`@ConditionalOn...`** annotations. These annotations act as checks to determine if the configuration should be applied. Key checks include:
    - **`@ConditionalOnClass`:** Checks if a specific class is present on the classpath (e.g., if you added the Hibernate library).
    - **`@ConditionalOnMissingClass`:** Checks if a specific class is *not* present.
    - **`@ConditionalOnBean`:** Checks if a specific bean has *already* been defined by the user.
    - **`@ConditionalOnMissingBean`:** Checks if a specific bean has *not* been defined by the user.
    - **`@ConditionalOnProperty`:** Checks if a specific configuration property is set to a certain value (e.g., `spring.datasource.url`).
3. **Applying Configuration:** If **all** the conditions for an auto-configuration class are met, Spring executes the methods inside, instantiating the required beans and adding them to the IoC container.
    - **Example:** When Spring finds the `spring-boot-starter-web` dependency:
        - It sees the **`WebMvcAutoConfiguration`** class.
        - The conditions check if the Tomcat server and Spring MVC classes are present (`@ConditionalOnClass`).
        - If true, it automatically configures an **embedded Tomcat server**, a `DispatcherServlet`, and the default view resolvers.

2. The Role of Starters (Opinionated Defaults)

The concept of auto-configuration is tightly coupled with **Starter Dependencies** (e.g., `spring-boot-starter-data-jpa`, `spring-boot-starter-web`).

- A Starter is essentially an empty JAR file that acts as a convenience tool.
- It bundles all the necessary, version-compatible dependencies required for a feature (e.g., Spring Web MVC, Tomcat, and Jackson for JSON).
- By adding a Starter, you put the necessary classes on the classpath, which then **triggers** the corresponding auto-configuration logic in Spring Boot.

3. Maintaining Control

Despite the automation, Spring Boot maintains the principle of "Developer Wins":

- **Overriding:** If you define your own custom bean for a specific component, Spring's auto-configuration will **back off** (due to the `@ConditionalOnMissingBean` check) and use your definition instead.
- **Excluding:** You can explicitly exclude specific auto-configuration classes by adding them to the `exclude` attribute of the `@SpringBootApplication` annotation if you never want a particular auto-configuration to run.

No, classes annotated with **`@Component`** (or its specializations like `@Service`, `@Repository`, and `@Controller`) are not considered "auto-configured" in the same sense as Spring Boot's automatic configuration mechanism.

Here is the breakdown of the distinction:

---

1. `@Component` Classes are Component-Scanned 🔎

- **What they are:** A class with `@Component` is an explicit, user-defined application component. You wrote the class, and you manually placed the annotation on it.
- **The Mechanism:** When the Spring application starts, the **Component Scan** feature (triggered by `@ComponentScan` in the main class) finds these classes, instantiates them, and registers them as beans in the IoC container.
- **Developer Control:** The developer has complete control over these beans' existence and implementation. Spring just handles the housekeeping (instantiation, DI, lifecycle).

---

2. Auto-Configuration Beans are Conditionally Configured ⚙️

- **What they are:** These are beans defined inside the Spring Boot framework itself, typically within classes annotated with **`@Configuration`** and specialized **`@ConditionalOn...`** annotations.
- **The Mechanism:** Spring Boot's **Auto-Configuration** mechanism checks runtime conditions (like the presence of a library or a user-defined property). If the conditions are met, it automatically creates and registers infrastructure beans (like an embedded Tomcat server, a database connection pool, or a specific JSON converter).
- **Developer Control:** The developer rarely sees or interacts with these configuration classes. Auto-configuration happens only if the developer **has not already provided their own version** of that bean (`@ConditionalOnMissingBean`).

Feature	`@Component` Classes	Auto-Configuration Classes
Triggered by	**`@ComponentScan`**	**`@EnableAutoConfiguration`** (or `@SpringBootApplication`)
Purpose	To define **application-specific** business logic components.	To configure **framework and infrastructure** components.
Default Logic	Always registered as a bean (if the package is scanned).	Registered only if certain **conditional checks** pass.

Life Cycle of Beans

Step1 - inversion control scan the components ,
Step2 - Load the class definition
Step3 - create the instance
Step4 - initialise the instance
Step5- use of instance
Step6 - Destroy of beans

The Spring Bean lifecycle and scopes are core concepts that define **when** a bean is created, **how** it is configured, and **how many** instances exist within the Spring IoC container.

1. Spring Bean Scopes 🔭

A bean's scope determines its visibility and how many instances of the object are created by the container.

Scope	Description	Behavior	Default ?
Singleton	**One instance** per Spring IoC container.	The container creates the bean when the application starts (eagerly) and reuses that single instance every time it is requested. This is ideal for services and repositories.	**Yes**
Prototype	**A new instance** is created every time the bean is requested.	The container manages the creation and dependency injection, but once created, it does **not manage the destruction** of the bean. This is suitable for stateful beans.	No
Request	**One instance** per HTTP request.	A new bean instance is created for each incoming web request and destroyed when the request completes. Only used in web applications.	No
Session	**One instance** per HTTP session.	A single bean instance is created for the entire duration of a user's session. Only used in web applications.	No
Application	**One instance** per web `ServletContext`.	Similar to a singleton but scoped to the entire web application context, not just the Spring container.	No

Export to Sheets

You define a bean's scope using the `@Scope` annotation (e.g., `@Scope("prototype")`).

---

2. Spring Bean Lifecycle 🔄

The lifecycle refers to the sequence of steps the IoC container performs from a bean's creation until its eventual destruction. This allows you to hook into the process to perform custom initialization or cleanup.

A. Container Initialization Phase (The "Birth")

1. **Instantiation:** The container finds the bean definition (via `@Component`, `@Service`, or `@Bean` method) and creates an instance of the class (e.g., calling the constructor).
2. **Populate Properties (DI):** The container injects the bean's dependencies (other beans) using **Dependency Injection** (Constructor, Setter, or Field Injection).
3. **Name/Context Awareness:** If the bean implements certain `*Aware` interfaces (like `BeanNameAware` or `ApplicationContextAware`), the container provides the necessary resources to the bean.
4. **`BeanPostProcessor.postProcessBeforeInitialization()`:** Custom post-processors get a chance to modify the bean before initialization methods run.
5. **Initialization Callbacks:** The container runs custom initialization code:
    - **`@PostConstruct`** method (a Java annotation).
    - `afterPropertiesSet()` method (if the bean implements the `InitializingBean` interface).
    - A custom method specified in the configuration (`init-method`).
6. **`BeanPostProcessor.postProcessAfterInitialization()`:** Custom post-processors get a final chance to wrap or proxy the bean (e.g., Spring AOP often wraps the bean here).

B. Bean Usage Phase

The bean is now fully initialized, configured, and ready for use by the application. This is the longest phase of its life.

C. Container Destruction Phase (The "Death")

1. **`@PreDestroy`** **Callback:** Before the container is shut down, it calls the bean's custom destruction code:
    - **`@PreDestroy`** method (a Java annotation).
    - `destroy()` method (if the bean implements the `DisposableBean` interface).
    - A custom method specified in the configuration (`destroy-method`).
2. **Destruction:** The bean instance is removed from the container and eventually garbage-collected.

Note: The destruction phase (steps C1-C2) is **only executed for beans with the Singleton scope**. The container does not track the destruction of **Prototype** beans.

Spring Boot manages configuration files through a highly flexible system known as **Externalized Configuration**, which ranks properties from different sources (including `application.properties` and YAML files) and merges them into a single, unified environment.

Here is how Spring Boot manages and consumes these files:

---

1. The Unified `Environment` Abstraction

Spring Boot loads all configuration settings into a single object, the **`Environment`** (part of the IoC container). This object doesn't care if the property came from a file, an environment variable, or a command-line argument—it treats them all as properties.

2. Configuration File Loading

The core of file management is based on the **`ApplicationContext`** (the IoC Container) searching for `application.properties` or `application.yml` files in specific locations:

1. **Classpath Root:** The highest priority default location is the `src/main/resources` folder.
2. **External Directory:** Spring Boot also looks outside the packaged JAR (e.g., in a `/config` folder next to the executable JAR), allowing operations teams to change settings without recompiling.

The process loads the file, parses the key-value pairs (for `.properties`) or the hierarchical structure (for `.yml`), and adds them to the **`Environment`**.

3. Profile-Specific Files (Conditional Loading)

Spring Boot makes it easy to manage configurations for different environments (like `dev`, `test`, or `prod`) using **Active Profiles**:

- **Default File:** The main `application.properties` (or `.yml`) loads first and establishes the baseline settings for *all* profiles.
- **Profile-Specific Overrides:** If the `prod` profile is active (e.g., set via `--spring.profiles.active=prod`), Spring Boot will then load a profile-specific file like **`application-prod.properties`** or **`application-prod.yml`**.
- **Merging:** The properties in the profile-specific file **override** the default properties. For instance, a `server.port` defined in `application-prod.yml` will replace the one in the main `application.yml`.

4. Precedence Hierarchy (The Configuration Order)

The most important aspect of configuration management is that Spring Boot doesn't just use files; it sources properties from **17 different locations** (levels) and prioritizes them.

When an application property is requested (e.g., the value for `spring.datasource.url`), Spring Boot checks the sources in a defined order, with items higher up the list **overriding** those below them:

Priority Level (Higher overrides Lower)	Example Source
Highest	**Command-line arguments** (e.g., `--server.port=9090`)
High	**Environment variables** (e.g., `SERVER_PORT=9090`)
Mid	**Profile-specific files** (`application-prod.yml`)
Low	**Default files** (`application.yml` in `resources`)
Lowest	**Default properties** defined inside Spring Boot

Here is the step-by-step flow of how the Spring IoC Container manages and loads configuration properties when the application starts, specifically focusing on `application.properties` and YAML files.

The Spring Boot Configuration Loading Flow

The entire process is orchestrated by the **`SpringApplication`** class, which creates and prepares the `ApplicationContext` (the IoC Container).

Phase 1: Environment Preparation (Sourcing Properties)

1. Create the Environment: The first thing Spring Boot does is create a central **`Environment`** object. This is a collection of key/value pairs that will eventually hold every single configuration setting for the entire application.

2. Load External Sources (Hierarchy Check): Spring Boot starts scanning all 17 supported locations for configuration, prioritizing them in a fixed order (from external/runtime properties down to internal/default ones).

- **Command Line & OS Variables:** Properties passed via the command line (`--server.port=9090`) and operating system environment variables (e.g., `SERVER_PORT`) are loaded first. **These hold the highest priority.**
- **Default File Search:** Spring searches for the common configuration files in standard locations (e.g., the `src/main/resources` folder and external `/config` folders).

    - It checks for **`application.properties`** or **`application.yml`**. The YAML format is processed to flatten its hierarchy into dot-separated properties before being added to the `Environment`.

3. Profile Resolution and Loading:

- Spring determines the **Active Profiles** (e.g., from the command line, `spring.profiles.active=prod`).
- It then searches for and loads any corresponding profile-specific files (e.g., `application-prod.yml`).
- These profile-specific properties are added to the `Environment` and automatically **override** any identical keys loaded from the default configuration files.

Phase 2: Bean Configuration and Wiring

4. Auto-Configuration Evaluation: With the complete, unified `Environment` now finalized, the Auto-Configuration mechanism runs.

- Spring inspects the classpath for all Auto-Configuration classes.
- It executes the `@ConditionalOnProperty` and related checks, using the values now available in the merged `Environment`.
- If a check passes, the relevant infrastructure beans (like `DataSource` or embedded server definitions) are created and configured using the properties it just loaded (e.g., using `spring.datasource.url`).

5. `@ConfigurationProperties` Binding: The property values in the `Environment` are now bound to custom user-defined configuration classes (Java objects annotated with `@ConfigurationProperties`).

- Example: The `spring.datasource.url` from the config file is automatically mapped to a specific field in a `DataSourceProperties` bean.

6. Dependency Injection: The IoC Container uses the newly created and configured infrastructure beans (from step 4) and custom application beans (from `@Component` scanning) to perform **Dependency Injection**, wiring the entire application together.

Outcome: By the time the application is fully started, every component is configured with the correct settings, prioritized according to the externalized configuration hierarchy.