

Spring Boot Core

1. What is Spring Boot?

Definition:

Spring Boot is an **open-source, Java-based framework** built on top of the **Spring Framework** that is designed to make **application development faster and easier** by eliminating boilerplate configurations.

It follows the principle of "**Convention over Configuration**", meaning you don't have to manually configure everything—Spring Boot provides sensible defaults.

Key Idea:

- Instead of writing hundreds of XML configuration lines (like in traditional Spring), Spring Boot automatically sets up most things using **auto-configuration**.
- You can focus on **business logic** instead of repetitive setup.

Example Analogy:

Think of **Spring Framework** as a "kitchen with all raw ingredients and utensils", and **Spring Boot** as a "kitchen where ingredients are pre-prepped, utensils are set, and you can start cooking right away".

Core Features:

- **Standalone Applications** – No need for an external server; runs with **embedded Tomcat/Jetty**.
- **Auto Configuration** – Automatically configures beans & settings based on dependencies.
- **Production Ready** – Metrics, health checks, and monitoring out of the box.
- **Minimal XML** – Most configurations are in application.properties or application.yml.
- **Dependency Management** – Uses **Spring Boot Starter** dependencies to simplify adding required libraries.

2. Why Use Spring Boot – Advantages & Use Cases

Advantages

1. Fast Development

- Rapid application development with minimal configuration.
- Default settings and prebuilt features save time.

2. Embedded Servers

- Built-in **Tomcat**, **Jetty**, or **Undertow** so no need to deploy WAR files manually.

3. Auto Configuration

- Detects and configures beans automatically based on the libraries you use.

4. Starter Dependencies

- One dependency (e.g., spring-boot-starter-web) pulls all required libraries.

5. Production-Ready Features

- Built-in health checks, metrics, logging, and application monitoring.

6. Microservices Friendly

- Ideal for **microservices architecture** with Spring Cloud integration.

7. No XML Configuration

- Mostly annotation-driven (@SpringBootApplication, @RestController, etc.).

8. Wide Community & Ecosystem

- Backed by Pivotal/VMware with vast community support.

Use Cases

1. REST APIs and Web Applications

- Build backend APIs quickly for mobile or web applications.

2. Microservices Architecture

- Spring Boot + Spring Cloud = powerful, scalable, distributed services.

3. Enterprise Applications

- Large-scale systems with built-in security, transaction management, and monitoring.

4. Event-Driven Systems

- With Spring Kafka, Spring AMQP, etc., for messaging-based solutions.

5. Batch Processing

- Automated job scheduling and batch jobs with **Spring Batch**.

6. Cloud-Native Applications

- Works seamlessly with AWS, Azure, GCP, Kubernetes, and Docker.

3. Features of Spring Boot

Spring Boot comes with a rich set of features that make Java application development **faster, easier, and more production-ready**.

1. Auto-Configuration

- Automatically configures Spring Beans based on the libraries in the project classpath.
 - Example:
 - If spring-boot-starter-web is present, it configures **Tomcat, Spring MVC**, and JSON converters automatically.
 - Eliminates the need for manual XML or Java-based configuration.
-

2. Standalone Applications

- Runs as a **standalone Java application** without needing an external application server.
- Uses **embedded servers** (Tomcat, Jetty, or Undertow).
- Run directly using:

```
java -jar myapp.jar
```

- No WAR deployment to servers like Tomcat manually.
-

3. Spring Boot Starters

- Special curated **dependency descriptors** that group commonly used dependencies for a particular functionality.
- Example:

- spring-boot-starter-web → Spring MVC, Jackson, Tomcat, and logging libraries.
 - spring-boot-starter-data-jpa → JPA + Hibernate + Database connectors.
 - Saves time by avoiding manual dependency version management.
-

4. Spring Boot CLI (Command Line Interface)

- Allows you to run and test Spring Boot applications quickly using Groovy scripts.
- Ideal for prototyping without full setup.
- Example:

```
spring run app.groovy
```

5. Production-Ready Features (Actuator)

- The **Spring Boot Actuator** module provides:
 - Health checks (/actuator/health)
 - Application metrics (CPU, memory usage)
 - Environment properties
 - Logging levels at runtime
 - Useful for monitoring and managing applications in production.
-

6. No XML Configuration

- Entirely **annotation-driven**.
- Uses @SpringBootApplication, @Configuration, @EnableAutoConfiguration, and @ComponentScan instead of large XML files.
- Example:

```
@SpringBootApplication  
public class MyApp { ... }
```

7. Layered Architecture Support

- Follows a clean architecture:
 - **Controller Layer** – Handles HTTP requests.
 - **Service Layer** – Business logic.
 - **Repository Layer** – Data access.
 - **Model Layer** – Data structures.
 - Promotes separation of concerns.
-

8. Externalized Configuration

- Configuration can be stored in:
 - application.properties
 - application.yml
 - Environment variables
 - Command-line arguments
- Example (application.properties):

```
server.port=9090  
spring.datasource.url=jdbc:mysql://localhost:3306/mydb
```

9. DevTools for Hot Reload

- **Spring Boot DevTools** automatically restarts the application when code changes are detected.
 - Speeds up development and testing.
-

10. Easy Integration with Other Technologies

- Works seamlessly with:
 - Spring Security
 - Spring Data
 - Spring Cloud
 - Messaging tools (Kafka, RabbitMQ)
 - Databases (MySQL, PostgreSQL, MongoDB)

11. Microservices Ready

- Provides everything required to build, run, and monitor **microservices**.
 - Supports distributed configuration, service discovery, and API gateways via Spring Cloud.
-

12. Logging Support

- Built-in logging with **Logback**.
- Can be customized via properties:

```
logging.level.org.springframework=DEBUG  
logging.file.name=app.log
```

In short: Spring Boot's features aim to **reduce boilerplate, speed up development, and make applications production-ready** out of the box.

4. Spring vs Spring Boot – Key Differences

Spring Boot is **built on top of Spring Framework** but adds extra features for faster development.

Aspect	Spring Framework	Spring Boot
Definition	A comprehensive framework for Java application development providing dependency injection, AOP, transaction management, etc.	An extension of Spring that simplifies setup, configuration, and deployment with auto-configuration and embedded servers.
Setup & Configuration	Requires manual setup (XML or Java-based configuration).	Auto-configuration; minimal manual setup needed.
Server Management	Requires an external server (Tomcat, Jetty, etc.) to deploy WAR files.	Comes with embedded servers (Tomcat, Jetty, Undertow) — runs as a standalone application.
Dependency Management	Developers must specify each dependency and its version.	Uses Spring Boot Starters to group and manage dependencies

Aspect	Spring Framework	Spring Boot
		automatically.
Project Structure	Requires more boilerplate code and configurations.	Opinionated defaults with ready-to-use project structure.
Deployment	Deploys on application servers like Tomcat, WebLogic, GlassFish, etc.	Deploys directly as a JAR with an embedded server using java -jar.
Learning Curve	Steeper — requires more understanding of configurations.	Easier for beginners due to convention over configuration.
Production-Ready Tools	No built-in production features; need manual integration.	Comes with Spring Boot Actuator for health checks, metrics, and monitoring.
Use Case	Suitable for complex, enterprise applications where fine-grained control over configuration is required.	Ideal for microservices, REST APIs, and quick application development.
Example Startup	Configure DispatcherServlet, ViewResolver, DataSource manually.	Just annotate with @SpringBootApplication and run.

Quick Analogy

- **Spring** = A fully stocked **kitchen** where you must set up everything (stove, utensils, ingredients).
 - **Spring Boot** = A **ready-to-cook meal kit** — most things are prepped; you just add final touches and start cooking.
-

Key Takeaway:

Spring Boot is **not a replacement** for Spring — it's a **tool to make Spring development faster, easier, and production-ready**.

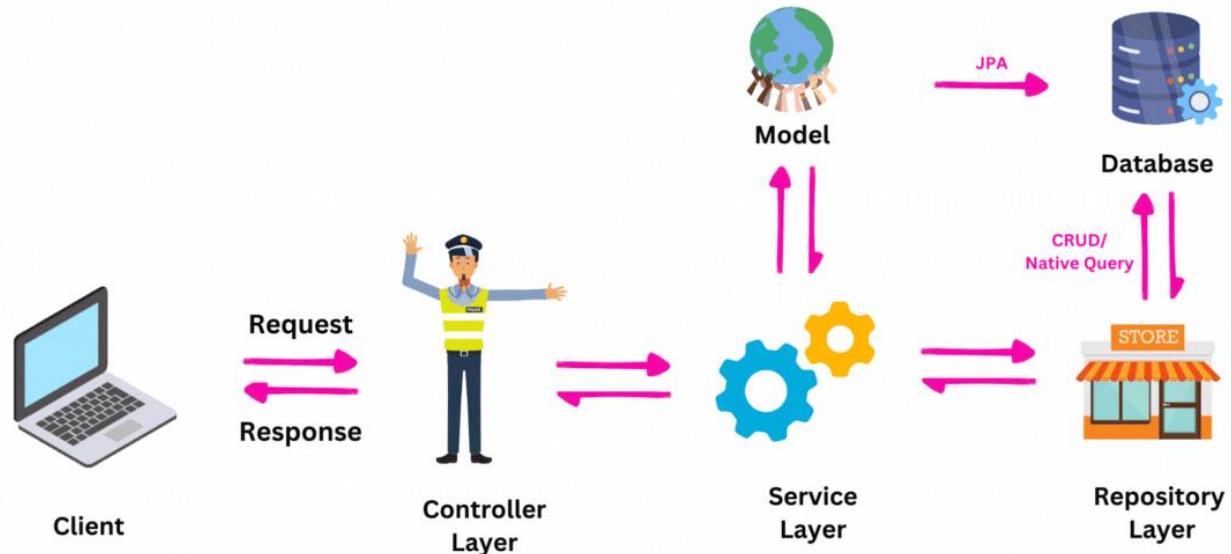
5. Spring Boot Architecture

Spring Boot's architecture builds on the **Spring Framework** but adds extra layers for **auto-configuration, embedded servers, and production-ready features**. It follows a layered structure where each layer plays a specific role.

High-Level Overview

Main Components:

1. **Client Layer** – User or application that sends requests.
2. **Server Layer** – Embedded server that receives and processes requests.
3. **Spring Container** – Core of Spring that manages beans, dependency injection, and configurations.
4. **Auto-Configuration** – Automatically sets up beans and services based on classpath dependencies.
5. **Application Layer** – Your business logic (Controllers, Services, Repositories).



1. Client Layer

- This is where the **request originates**.
- Can be:
 - Web browser

- Mobile app
 - API consumer (Postman, Curl, another microservice)
 - Sends HTTP requests to the application (e.g., GET /users).
-

2. Server Layer (Embedded Server)

- Spring Boot runs with an **embedded server** like:
 - **Tomcat** (default)
 - Jetty
 - Undertow
 - This server listens for HTTP requests and forwards them to the Spring DispatcherServlet.
 - No need for manual deployment to an external server.
-

3. Spring Container

- Core part of **Spring Framework**.
- Responsible for:
 - Creating and managing beans.
 - Handling **Dependency Injection (DI)**.
 - Managing application lifecycle.
- Uses **IoC (Inversion of Control)** to provide required dependencies automatically.

Key modules involved:

- **DispatcherServlet** – Front controller that routes requests to controllers.
 - **Handler Mapping** – Finds the correct controller method for a given URL.
 - **View Resolver** – Decides how to render the response (JSON, HTML, etc.).
-

4. Auto-Configuration

- Spring Boot **automatically configures** the application based on:
 1. **Classpath dependencies** – e.g., if spring-boot-starter-data-jpa is present, it sets up JPA and Hibernate automatically.

2. **Property settings** in application.properties or application.yml.

- Uses the @EnableAutoConfiguration annotation (part of @SpringBootApplication).
 - If you want to override defaults, you can provide **custom configuration**.
-

5. How the Flow Works

1. **Client** sends a request (e.g., GET /hello).
 2. **Embedded Server** (Tomcat) receives it.
 3. Request is passed to **DispatcherServlet** in the **Spring Container**.
 4. **Handler Mapping** finds the right Controller method.
 5. Controller calls **Service Layer** for business logic.
 6. Service calls **Repository Layer** to interact with the database.
 7. Response goes back the same route to the **Client**.
-

Key Benefits of this Architecture

- **Loose Coupling** – Components are independent.
 - **Scalability** – Suitable for microservices.
 - **Flexibility** – Easy to replace beans or configs.
 - **Minimal Configuration** – Thanks to auto-configuration.
-

In short:

Spring Boot architecture combines **Spring's IoC container** with **auto-configuration** and **embedded servers**, making it possible to build and run production-ready applications with minimal setup.

6. Creating Spring Boot Applications

Spring Boot applications can be created **quickly and easily** using two main approaches:

1. **Using Spring Initializr Website**
 2. **Using IDE Tools (IntelliJ IDEA, Spring Tool Suite, Eclipse)**
-

1. Using Spring Initializr (Website)

Spring Initializr is an online project generator that helps you create a ready-to-use Spring Boot project structure in a few clicks.

Steps:

1. **Open Website**
Visit <https://start.spring.io>
2. **Choose Project Type**
 - o **Maven Project** (most common) or **Gradle Project**
3. **Select Language**
 - o Java (default), Kotlin, or Groovy
4. **Spring Boot Version**
 - o Choose the latest **stable** release.
5. **Project Metadata**
 - o **Group**: Reverse domain name (e.g., com.example)
 - o **Artifact**: Project name (e.g., demo)
 - o **Name**: Application name
 - o **Package Name**: Defaults to group + artifact
6. **Packaging & Java Version**
 - o Packaging: **Jar** (default) or **War**
 - o Java: Select your installed Java version (e.g., 17)
7. **Add Dependencies**
 - o Click **Add Dependencies** and choose relevant starters.
Example for REST API:
 - Spring Web
 - Spring Boot DevTools (for hot reload)
 - Spring Data JPA
 - MySQL Driver

8. Generate Project

- Click **Generate** to download a .zip file.
 - Extract it and open in your IDE.
-

Advantages of using Spring Initializr:

- No manual folder structure creation.
 - Automatically adds correct dependencies and versions.
 - Works with Maven and Gradle.
-

2. Using an IDE (IntelliJ IDEA, STS, Eclipse)

Most modern Java IDEs have **Spring Initializr integration** built in.

a) IntelliJ IDEA (Ultimate / Community with Plugin)

1. File → New → Project
2. Select **Spring Initializr**.
3. Choose SDK (Java version).
4. Fill **Project Metadata** (Group, Artifact, etc.).
5. Select **Dependencies** from built-in list.
6. Finish → IntelliJ downloads and sets up the project.

7. Internal Flow of a Spring Boot Application

Spring Boot's internal working is based on **Spring Framework's core concepts** (IoC, DI, DispatcherServlet) but adds **auto-configuration, embedded server, and starters** to simplify the process.

Step-by-Step Flow

1. Application Startup

- The application starts with a **main class** annotated with:

```
@SpringBootApplication  
public class MyApp {  
    public static void main(String[] args) {  
        SpringApplication.run(MyApp.class, args);  
    }  
}
```

- @SpringBootApplication** is a **meta-annotation** that combines:
 - @Configuration** – Marks class as a source of bean definitions.
 - @EnableAutoConfiguration** – Enables Spring Boot's auto-config.
 - @ComponentScan** – Scans for components in the package.
- SpringApplication.run()** triggers:
 - Bootstrapping** – Creates an ApplicationContext (Spring Container).
 - Loads Beans** – Scans and registers beans in IoC container.
 - Applies Auto-Configuration** based on dependencies.

2. Auto-Configuration Phase

- Enabled by **@EnableAutoConfiguration**.
- Uses **spring.factories** files from dependencies (inside JARs) to check which auto-configurations to apply.
- Example:
 - If **spring-boot-starter-web** is present → Configures DispatcherServlet, Tomcat, JSON converters automatically.
 - If **spring-boot-starter-data-jpa** is present → Configures Hibernate, DataSource, JPA repositories.

3. Embedded Server Initialization

- Based on dependencies, Spring Boot starts:
 - Tomcat** (default)
 - Jetty / Undertow (if added)
- Runs within the same JVM, so the app can be started with:

```
java -jar myapp.jar
```

- No WAR deployment needed.
-

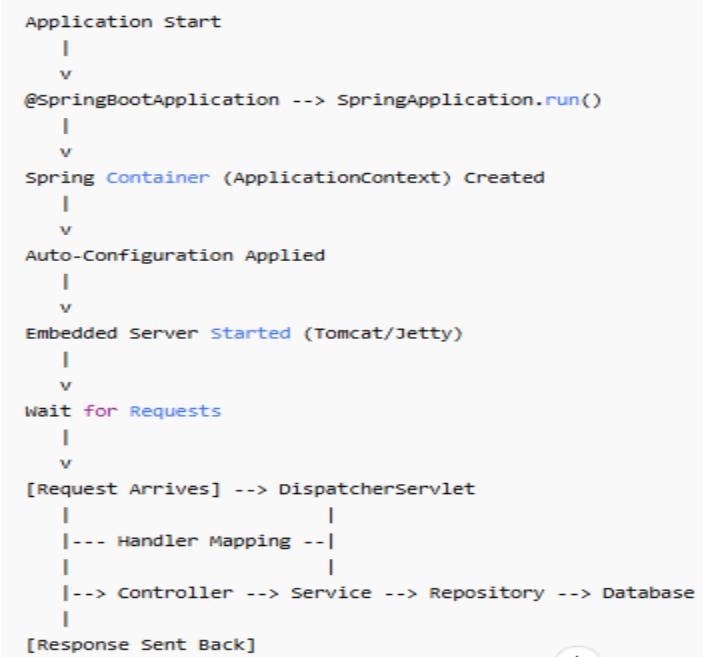
4. Request Handling (Spring MVC Flow)

When a client sends an HTTP request:

1. **Embedded Server** receives the request.
 2. **DispatcherServlet** (Front Controller) processes it.
 3. **Handler Mapping** finds the appropriate controller method based on URL and HTTP method.
 4. The **Controller** executes and calls:
 - **Service Layer** for business logic.
 - **Repository Layer** for database interaction (if needed).
 5. The response is returned as:
 - JSON/XML (via @RestController or @ResponseBody)
 - HTML page (via templates like Thymeleaf)
-

5. Response Generation

- **View Resolver** decides how to render the response:
 - JSON for APIs
 - HTML for web pages
- Response is sent back to the **Client** through the embedded server.



□ In short:

Spring Boot starts by **bootstrapping the Spring container**, applies **auto-configuration** based on dependencies, starts an **embedded server**, and uses the **Spring MVC flow** to handle requests and return responses.

8. @SpringBootApplication Annotation

The `@SpringBootApplication` annotation is the **main entry point** for a Spring Boot application.

It is placed on the **main class** of your application to enable **auto-configuration, component scanning, and bean registration**.

Definition

`@SpringBootApplication` is a **meta-annotation** that combines three commonly used Spring annotations:

```
@SpringBootApplication  
public class MyApp {  
    public static void main(String[] args) {  
        SpringApplication.run(MyApp.class, args);  
    }  
}
```

```
}
```

It Equals To:

```
@Configuration  
@EnableAutoConfiguration  
@ComponentScan  
public class MyApp {  
    ...  
}
```

1. @Configuration

- Marks the class as a **configuration class** for Spring's IoC container.
- Allows you to define **beans** using @Bean methods.
- Example:

```
@Configuration  
public class MyConfig {  
    @Bean  
    public MyService myService() {  
        return new MyService();  
    }  
}
```

2. @EnableAutoConfiguration

- Tells Spring Boot to **automatically configure** the application based on dependencies present in the classpath.
- Uses the spring.factories mechanism to load configurations.
- Example:
 - If spring-boot-starter-web is present → Configures DispatcherServlet, Jackson, Tomcat automatically.
- You can **exclude** specific auto-configurations:

```
@SpringBootApplication(exclude = { DataSourceAutoConfiguration.class })
```

3. @ComponentScan

- Scans the **current package and sub-packages** for:
 - @Component
 - @Service
 - @Repository
 - @Controller
 - @RestController
- Registers them as Spring Beans in the IoC container.
- Example:

```
@Component  
public class MyBean { ... }
```

This bean will be discovered automatically.

Flow When Using @SpringBootApplication

1. Marks the class as a configuration class (@Configuration).
2. Triggers auto-configuration (@EnableAutoConfiguration).
3. Scans and registers all beans in the package (@ComponentScan).
4. Boots the application using SpringApplication.run()

```
@SpringBootApplication  
|  
+--> @Configuration + Bean definitions  
|  
+--> @EnableAutoConfiguration + Auto-setup based on dependencies  
|  
+--> @ComponentScan + Scan beans in package
```

□ Key Takeaway:

@SpringBootApplication **reduces boilerplate** by combining three essential Spring annotations into one, making it the standard way to start a Spring Boot application.

9. Spring Boot Auto-Configuration

Spring Boot **Auto-Configuration** is one of its **core features** that allows it to automatically configure your application **based on the dependencies available in the classpath** — without requiring manual configuration.

What is Auto-Configuration?

- Instead of manually defining beans, data sources, and configurations, Spring Boot **intelligently guesses and sets them up**.
 - It uses sensible **defaults** but lets you **override** them if needed.
-

Annotation: @EnableAutoConfiguration

- The main trigger for auto-configuration in Spring Boot.
 - **Included inside** @SpringBootApplication.
 - Instructs Spring Boot to **scan the classpath** and **load configuration classes** from META-INF/spring.factories.
-

How Auto-Configuration Works (Step-by-Step)

1. Classpath Scanning

- Spring Boot looks at the **dependencies** in your pom.xml or build.gradle.
- Example:
 - If spring-boot-starter-web is present → Configures Spring MVC, Jackson, and an embedded Tomcat server.
 - If spring-boot-starter-data-jpa is present → Configures Hibernate, JPA repositories, and a DataSource.

2. Find Auto-Configuration Classes

- Spring Boot loads a list of auto-configuration classes from:
 - META-INF/spring.factories
- Each dependency may contribute its own auto-configuration classes.

3. Conditional Configuration

- Each auto-configuration class is annotated with **conditional annotations**:
 - @ConditionalOnClass – Apply config only if a specific class is on the classpath.
 - @ConditionalOnMissingBean – Apply config only if a bean is not already defined by the user.
 - @ConditionalOnProperty – Apply config based on properties in application.properties.

4. Bean Creation

- Auto-configuration creates beans for components like:
 - DataSource
 - EntityManagerFactory
 - MessageConverters
 - ViewResolvers

5. User Overrides

- If you define a bean manually, **auto-config will back off**.
- Example:

```
@Bean  
public DataSource myCustomDataSource() {  
    return new HikariDataSource();  
}
```

→ Spring Boot won't create its default DataSource.

Example

If your project contains:

```
<dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-web</artifactId>  
</dependency>
```

Spring Boot will:

- Detect DispatcherServlet and Tomcat on the classpath.
- Automatically configure:

- Web server (Tomcat)
 - JSON support (Jackson)
 - Static resource handling
 - Error page handling
-

Disabling Auto-Configuration

You can disable specific configurations:

```
@SpringBootApplication(exclude = { DataSourceAutoConfiguration.class })  
public class MyApp { ... }
```

Key Conditional Annotations in Auto-Configuration

Annotation	Purpose
@ConditionalOnClass	Config only if a class is on the classpath
@ConditionalOnMissingBean	Config only if a bean is not already present
@ConditionalOnProperty	Config only if a property is set in application.properties
@ConditionalOnMissingClass	Config only if a class is not on the classpath

10. Difference between Autowiring and Auto-Configuration in Spring Boot

1. Autowiring

- **Definition:**
Autowiring is the **process of automatically injecting bean dependencies** into other beans by Spring's IoC (Inversion of Control) container.
- **Purpose:**
To avoid manual bean wiring using XML or explicit @Bean methods.
- **How it Works:**
 - Spring looks for a **matching bean** in the application context.
 - Injects it into the target class **based on type, name, or constructor**.
 - Controlled mainly via the @Autowired annotation.

- **Scope:**
Works **within the existing beans** in the Spring Application Context.
It **does not create beans**; it only injects already defined ones.
- **Example:**

```
@Service  
public class UserService {  
    @Autowired  
    private UserRepository userRepository;  
}
```

Here, Spring **finds** UserRepository bean and injects it into UserService.

2. Auto-Configuration

- **Definition:**
Auto-Configuration is the **process where Spring Boot automatically creates and configures beans** for you based on the classpath contents and properties.
- **Purpose:**
To reduce manual configuration by providing sensible defaults for many common scenarios.
- **How it Works:**
 - Triggered by the `@EnableAutoConfiguration` (inside `@SpringBootApplication`).
 - Uses META-INF/spring.factories to load configuration classes.
 - Checks for **specific classes on the classpath** and creates beans accordingly.
- **Scope:**
It **creates and registers beans** automatically **before** autowiring happens.
- **Example:**
If you add **Spring Web** dependency:
 - Auto-Configuration will create DispatcherServlet, DefaultErrorHandler, etc., without you writing a single bean definition.

3. Key Differences Table

Aspect	Autowiring	Auto-Configuration
Definition	Injects existing beans into other beans	Automatically creates and configures beans
Purpose	Dependency injection	Reduce boilerplate configuration
Trigger	@Autowired, @Inject	@EnableAutoConfiguration
Bean Creation?	No – only injects	Yes – creates beans if needed
When it Happens	After beans are created	Before autowiring
Customization	Choose which bean to inject	Override defaults with properties or custom beans

In short:

- **Auto-Configuration:** Decides *what beans to create* based on your project setup.
- **Autowiring:** Decides *where to inject* those beans inside your code.

11. Spring Boot Starters & Dependencies

1. What are Starters in Spring Boot?

- **Definition:**

Starters are **pre-defined dependency descriptors** provided by Spring Boot to simplify dependency management.

Instead of adding multiple dependencies manually in your pom.xml (Maven) or build.gradle (Gradle), you just include **one starter**, and it will pull all the required libraries for that module.

- **Why Starters?**

- Avoids **dependency hell** (version mismatches).
- Reduces boilerplate setup.
- Brings **opinionated defaults** so you don't have to manually configure libraries.

2. Naming Convention

- All Spring Boot starters follow the pattern:
 - `spring-boot-starter-<module>`
 - Examples:
 - `spring-boot-starter-web` → for building web applications.
 - `spring-boot-starter-data-jpa` → for JPA and Hibernate integration.
 - `spring-boot-starter-security` → for Spring Security.
-

3. Commonly Used Spring Boot Starters

Starter Name	Purpose
<code>spring-boot-starter</code>	Core starter including logging, YAML/JSON parsing, etc.
<code>spring-boot-starter-web</code>	For building web apps & REST APIs using Spring MVC + Tomcat (default) or Jetty/Undertow.
<code>spring-boot-starter-data-jpa</code>	For database interaction using JPA (Hibernate).
<code>spring-boot-starter-thymeleaf</code>	For server-side HTML rendering with Thymeleaf template engine.
<code>spring-boot-starter-security</code>	Adds Spring Security for authentication & authorization.
<code>spring-boot-starter-test</code>	Brings JUnit, Mockito, Spring Test for unit & integration testing.
<code>spring-boot-starter-mail</code>	For sending emails using JavaMail.
<code>spring-boot-starter-actuator</code>	For monitoring & management endpoints (health, metrics, etc.).
<code>spring-boot-starter-batch</code>	For batch processing jobs.
<code>spring-boot-starter-validation</code>	Adds Hibernate Validator for bean validation.

4. Example – Adding Starter in Maven

```
<dependencies>
```

```
<!-- For REST APIs -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<!-- For Database -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<!-- For Testing -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
</dependencies>
```

5. How Starters Work Internally

- Starters are **just dependency groups** declared in their own pom.xml.
- They **don't contain actual code** but **reference the real libraries**.
- Spring Boot uses **Spring Boot Dependency Management** to ensure **version compatibility** between these libraries.

Example:

spring-boot-starter-web includes:

- spring-web
 - spring-webmvc
 - jackson-databind (for JSON)
 - spring-boot-starter-tomcat
 - spring-boot-starter-validation
-

6. Benefits of Spring Boot Starters

- Easy dependency management
- No need to manually search for compatible versions
- Follows opinionated defaults for faster development
- Reduces boilerplate pom.xml clutter

12. Stereotype Annotations in Spring Boot

Stereotype annotations are **special annotations** provided by Spring Framework to **mark classes as specific Spring-managed components** in the application context. They help **Spring automatically detect and register beans** during component scanning.

1. @Component

- **Definition:** Generic stereotype annotation that marks a class as a Spring-managed bean.
- **Purpose:** Used for any **generic component** that doesn't fit into a specific role (service, repository, etc.).
- **Where to use:** Utility classes, helper classes, or any bean you want Spring to manage.
- **Example:**

```
import org.springframework.stereotype.Component;
```

```
@Component
public class EmailValidator {
    public boolean isValid(String email) {
        return email.contains("@");
    }
}
```

Here, EmailValidator will be detected and registered as a **bean** automatically.

2. @Service

- **Definition:** Specialized version of @Component used for **service layer** classes.
- **Purpose:** Indicates that the class contains **business logic**.
- **Where to use:** Classes that perform calculations, processing, or coordinate business rules.
- **Example:**

```
import org.springframework.stereotype.Service;
```

```
@Service
public class PaymentService {
    public void processPayment(double amount) {
        System.out.println("Processing payment of " + amount);
    }
}
```

- Internally, @Service works like @Component but is used for **semantic clarity**.
-

3. @Repository

- **Definition:** Specialized @Component for **data access layer** (DAO classes).
- **Purpose:**
 - Marks classes that interact with the **database**.
 - Adds **exception translation**: converts database exceptions into **Spring's DataAccessException**.
- **Where to use:** JDBC, JPA, Hibernate repository classes.
- **Example:**

```
import org.springframework.stereotype.Repository;
```

```
@Repository
public class UserRepository {
    public String findUserById(int id) {
        return "User with ID: " + id;
    }
}
```

}

4. @Controller

- **Definition:** Specialized @Component for **MVC web controllers**.
- **Purpose:** Handles **web requests** and returns **views (HTML pages)**.
- **Where to use:** Spring MVC applications (not REST APIs).
- **Example:**

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class HomeController {
    @GetMapping("/")
    public String homePage() {
        return "index"; // returns view name
    }
}
```

Here, "index" refers to a **template file** like index.html.

5. @RestController

- **Definition:** Combination of @Controller + @ResponseBody.
- **Purpose:** Used for **REST APIs**.
Automatically serializes Java objects to **JSON/XML** in the response.
- **Where to use:** REST API endpoints.
- **Example:**

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
```

```
@RestController
public class ProductController {
    @GetMapping("/products")
```

```
public String getProducts() {  
    return "List of products";  
}  
}
```

- Here, "List of products" will be sent as **raw text/JSON**, not as a view name.
-

Summary Table

Annotation	Layer / Purpose	Special Feature
@Component	Generic bean	Basic Spring bean
@Service	Service layer	Business logic clarity
@Repository	DAO layer	Exception translation
@Controller	MVC controller	Returns view name
@RestController	REST API	JSON/XML response

□ Key Points to Remember

- All these annotations are **detected via Component Scanning** (enabled by `@ComponentScan`).
- They make the code **more readable** by clearly indicating the role of each class.
- `@RestController` is for **APIs**, while `@Controller` is for **web pages**.

13. Dependency Injection in Spring Boot

1. What is Dependency Injection (DI)?

- **Definition:** A design pattern where the dependencies (objects a class needs) are provided externally rather than the class creating them itself.
- **Purpose in Spring Boot:**
 - Promotes loose coupling between classes.
 - Makes code easier to test, maintain, and extend.
- Spring's IoC (Inversion of Control) container manages DI by creating and injecting beans automatically.

Types of Dependency Injection in Spring Boot

Spring Boot supports **three main types**:

1. Constructor Injection (Recommended)

Definition: Dependencies are passed via the class constructor.

Example:

```
import org.springframework.stereotype.Component;
```

```
@Component
public class Car {
    private final Engine engine;

    // Constructor Injection
    public Car(Engine engine) {
        this.engine = engine;
    }

    public void start() {
        engine.run();
    }
}
```

Why Constructor Injection is Recommended:

- Makes dependencies **immutable** (use final keyword).
 - Ensures dependency is not null at runtime.
 - Works well with testing frameworks.
 - Detects missing dependencies at compile-time.
-

2.Setter Injection

Definition: Dependencies are passed through setter methods after the object is created.

Example:

```
import org.springframework.stereotype.Component;
```

```
@Component
public class Car {
    private Engine engine;

    // Setter Injection
    public void setEngine(Engine engine) {
        this.engine = engine;
    }

    public void start() {
        engine.run();
    }
}
```

When to Use:

- For **optional** dependencies.
- When dependencies can change during the object's lifecycle.

3. Field Injection (Not Recommended for Production)

Definition: Dependencies are injected directly into the class fields using `@Autowired`.

Example:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
```

```
@Component
```

```
public class Car {  
  
    @Autowired  
    private Engine engine; // Field Injection  
  
    public void start() {  
        engine.run();  
    }  
}
```

Drawbacks:

- Makes testing harder (requires reflection to set private fields in tests).
- Hides dependencies (less clear what the class actually needs).
- Breaks immutability.

Best Practices

- **Use Constructor Injection** for **mandatory** dependencies.
- **Use Setter Injection** for **optional** dependencies.
- **Avoid Field Injection** in real-world applications — keep it only for quick prototypes or very small apps.

14. Configuration Files in Spring Boot

Spring Boot provides two main ways to configure your application:

1. **application.properties** – Key-value based configuration
2. **application.yml** – YAML (hierarchical) configuration

These files allow you to define:

- Server settings
- Database connection details
- Logging levels
- Custom application properties
- Third-party library configurations

By default, **Spring Boot automatically loads these files** from:

/src/main/resources/

1. application.properties

- Format: **key=value**
- Simple and widely used.
- Good for small configurations.

Example:

```
# Server configuration
server.port=8081
server.servlet.context-path=/myapp

# Database configuration
spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=root
spring.datasource.password=admin
spring.jpa.hibernate.ddl-auto=update

# Logging configuration
logging.level.org.springframework=DEBUG
```

2. application.yml

- Format: **YAML** (Yet Another Markup Language)
- Supports hierarchical/nested data.
- More readable for complex configurations.

Example:

```
server:
  port: 8081
  servlet:
    context-path: /myapp

spring:
  datasource:
    url: jdbc:mysql://localhost:3306/mydb
    username: root
```

```
password: admin
jpa:
  hibernate:
    ddl-auto: update

logging:
  level:
    org.springframework: DEBUG
```

application.properties vs application.yml

Feature	application.properties	application.yml
Format	Key-Value	Hierarchical
Readability	Less for large configs	More readable
Comments	#	#
Complex data	Harder to represent	Easy with nesting

Priority Order in Spring Boot

When both files exist:

1. application.properties
2. application.yml

The **last loaded** file overrides the previous values if keys are the same.

Profiles in Configuration Files

Spring Boot supports **profiles** to manage different environments (dev, test, prod).

Example:

```
# application-dev.properties
server.port=8082
```

```
# application-prod.properties
```

server.port=8080

Run the application with:

```
java -jar myapp.jar --spring.profiles.active=dev
```

Best Practices

- Use **application.yml** for complex configurations.
- Keep **secrets** (passwords, keys) in environment variables or external config.
- Use **profiles** to separate environment-specific settings.
- Avoid hardcoding values in code – always externalize in config files.

15. Profiles in Spring Boot

Profiles in Spring Boot allow you to define **different configurations** for **different environments** (e.g., development, testing, production) and activate them based on the environment where the application is running.

1. Why Use Profiles?

- Applications behave differently in different environments.
 - Example:
 - In **development**, you may use an in-memory H2 database.
 - In **production**, you may connect to MySQL/PostgreSQL.
 - Profiles help **avoid hardcoding environment-specific values**.
-

2. How Profiles Work

- You create **multiple configuration files** or beans.
 - You **tag them with a profile name** using @Profile.
 - At runtime, you **activate a profile** via application.properties, application.yml, or command line arguments.
-

3. Activating Profiles

a) In *application.properties*

```
spring.profiles.active=dev
```

b) In *application.yml*

```
spring:  
  profiles:  
    active: dev
```

c) Using Command Line

```
java -jar myapp.jar --spring.profiles.active=prod
```

d) Using Environment Variables

```
export SPRING_PROFILES_ACTIVE=prod
```

4. Profile-Specific Properties Files

You can create separate property files for each profile:

```
application.properties      # Common properties  
application-dev.properties # Dev-specific  
application-test.properties # Test-specific  
application-prod.properties # Prod-specific
```

When a profile is active, Spring Boot **automatically loads**:

- Common properties from application.properties
 - Environment-specific properties from application-{profile}.properties
-

5. Using @Profile Annotation

The `@Profile` annotation is used to **define beans** that should only load in a specific profile.

Example:

```
@Component
```

```
@Profile("dev")
public class DevDataSourceConfig {
    public DevDataSourceConfig() {
        System.out.println("Dev DataSource loaded");
    }
}

@Component
@Profile("prod")
public class ProdDataSourceConfig {
    public ProdDataSourceConfig() {
        System.out.println("Prod DataSource loaded");
    }
}
```

If the profile is dev, only DevDataSourceConfig will be loaded.

6. Multiple Profiles

You can activate **more than one profile**:

spring.profiles.active=dev,security

Or in YAML:

```
spring:
  profiles:
    active:
      - dev
      - security
```

7. Default Profile

- If no profile is set, **Spring Boot uses the default profile**.
- You can explicitly define:

spring.profiles.default=dev

8. Real-Life Example

application-dev.properties

```
spring.datasource.url=jdbc:h2:mem:devdb  
spring.datasource.username=devuser
```

application-prod.properties

```
spring.datasource.url=jdbc:mysql://localhost:3306/proddb  
spring.datasource.username=produser
```

Main Class

```
@SpringBootApplication  
public class ProfileDemoApplication {  
    public static void main(String[] args) {  
        SpringApplication.run(ProfileDemoApplication.class, args);  
    }  
}
```

If we run:

```
java -jar app.jar --spring.profiles.active=prod
```

→ The MySQL datasource will be used instead of H2.

9. Best Practices

- Keep **common configs** in application.properties.
- Use profiles for **environment-specific** settings only.
- Avoid **hardcoding environment variables** in code.
- Combine profiles when needed (e.g., dev + security).

16. Externalized Configuration & Property Injection (@Value, @ConfigurationProperties)

1. What is Externalized Configuration in Spring Boot?

Spring Boot allows you to **keep configuration values outside the source code** so that you can easily change them without modifying or recompiling your application.

- Useful for:
 - Changing database URLs
 - Switching environments (dev, test, prod)
 - Managing secrets (API keys, passwords)
 - Supported file formats:
 - application.properties
 - application.yml
 - Environment variables
 - Command-line arguments
 - External config files
 - Config server (Spring Cloud)
-

2. Property Injection

Spring Boot provides **two main ways** to inject configuration values into your application:

A. Using @Value Annotation

- Injects a single property value directly from configuration files, environment variables, or system properties.

Syntax:

```
@Value("${property.key.defaultValue}")  
private String value;
```

- \${property.key} → Reads the value of property.key
- :defaultValue → Optional default if property not found

Example:

application.properties

```
app.name=My Spring Boot App  
app.version=1.0.0
```

Java Code

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class AppConfig {

    @Value("${app.name}")
    private String appName;

    @Value("${app.version}")
    private String appVersion;

    @Value("${app.author:Unknown}") // Default value
    private String author;

    public void printConfig() {
        System.out.println("App Name: " + appName);
        System.out.println("Version: " + appVersion);
        System.out.println("Author: " + author);
    }
}
```

B. Using @ConfigurationProperties

- Binds **multiple related properties** into a Java POJO.
- Requires a prefix in properties file.
- **Better for structured data.**

Step 1 – Add Dependency

Spring Boot already includes spring-boot-configuration-processor for metadata hints (optional but recommended):

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional>
</dependency>
```

Step 2 – Properties File

```
app.name=My Spring Boot App  
app.version=1.0.0  
app.author=Prathamesh Jadhav  
app.features.auth=true  
app.features.payment=false
```

Step 3 – Java Class

```
import org.springframework.boot.context.properties.ConfigurationProperties;  
import org.springframework.stereotype.Component;  
  
import java.util.Map;  
  
@Component  
@ConfigurationProperties(prefix = "app")  
public class AppProperties {  
    private String name;  
    private String version;  
    private String author;  
    private Map<String, Boolean> features; // For nested values  
  
    // Getters & Setters  
    public String getName() { return name; }  
    public void setName(String name) { this.name = name; }  
  
    public String getVersion() { return version; }  
    public void setVersion(String version) { this.version = version; }  
  
    public String getAuthor() { return author; }  
    public void setAuthor(String author) { this.author = author; }  
  
    public Map<String, Boolean> getFeatures() { return features; }  
    public void setFeatures(Map<String, Boolean> features) { this.features =  
        features; }  
}
```

Step 4 – Using It

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class ConfigService {
    @Autowired
    private AppProperties appProperties;

    public void displayConfig() {
        System.out.println("App Name: " + appProperties.getName());
        System.out.println("Version: " + appProperties.getVersion());
        System.out.println("Author: " + appProperties.getAuthor());
        System.out.println("Features: " + appProperties.getFeatures());
    }
}
```

3. Externalizing Config in Multiple Ways

Spring Boot looks for configuration in **this order (highest to lowest priority)**:

1. Command-line arguments
 2. Java System properties (-Dproperty=value)
 3. OS environment variables
 4. Application properties inside application.properties or application.yml
 5. Default values in code (@Value("\${prop:default}"))
-

4. application.properties vs application.yml

Both serve the same purpose, but YAML is more structured and readable.

Example – application.yml

app:

```
name: My Spring Boot App
version: 1.0.0
author: Prathamesh Jadhav
features:
```

```
auth: true  
payment: false
```

17. Logging in Spring Boot (Default logging, SLF4J, Logback)

1. What is Logging in Spring Boot?

Logging is the process of recording runtime information of an application for:

- **Debugging** errors
- **Monitoring** system behavior
- **Auditing** actions
- **Performance analysis**

Spring Boot provides **default logging support** and allows integration with popular logging frameworks like:

- **Java Util Logging (JUL)**
- **Log4j 2**
- **Logback (default)**

2. Default Logging in Spring Boot

- By default Spring Boot uses **Logback** for logging.
- The logging facade used is **SLF4J** (Simple Logging Facade for Java).
- The default log pattern prints:
 - Date & time
 - Log level (INFO, DEBUG, ERROR, etc.)
 - Thread name
 - Logger name
 - Message

Example Default Log Output:

```
2025-08-14T10:15:30.123 INFO 12345 --- [ main]  
com.example.demo.DemoApplication : Started DemoApplication in 3.456 seconds
```

Log Levels in Spring Boot:

1. **TRACE** – Detailed low-level logs for debugging.
 2. **DEBUG** – Debugging information.
 3. **INFO** – General application flow.
 4. **WARN** – Warnings about potential problems.
 5. **ERROR** – Serious issues that need attention.
 6. **OFF** – Disable logging.
-

3. Changing Log Level

You can change log levels in **application.properties** or **application.yml**.

Example – application.properties

```
logging.level.root=INFO  
logging.level.com.example=DEBUG
```

Example – application.yml

```
logging:  
  level:  
    root: INFO  
    com.example: DEBUG
```

4. SLF4J (Logging Facade)

- **SLF4J** is a **facade** — it provides a common logging API but doesn't log by itself.
- Works with multiple logging frameworks (Logback, Log4j, etc.).
- You log using Logger from SLF4J, but the actual logging is done by the underlying framework.

Example:

```
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
import org.springframework.stereotype.Component;
```

```
@Component
public class MyService {
    private static final Logger logger = LoggerFactory.getLogger(MyService.class);

    public void processData() {
        logger.info("Processing started...");
        logger.debug("Debugging details here");
        logger.error("An error occurred");
    }
}
```

5. Logback (Default Implementation)

- **Logback** is the **default logging implementation** in Spring Boot.
- Supports:
 - Rolling logs (daily or size-based rotation)
 - Different formats for different environments
 - XML-based configuration

Custom Logback Configuration:

Create src/main/resources/logback-spring.xml:

```
<configuration>
    <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d{yyyy-MM-dd HH:mm:ss} %-5level %logger{36} -
            %msg%n</pattern>
        </encoder>
    </appender>

    <root level="INFO">
        <appender-ref ref="CONSOLE" />
    </root>
</configuration>
```

6. File Logging in Spring Boot

To write logs to a file, just set in application.properties:

```
logging.file.name=app.log  
logging.file.path=logs
```

This will create a logs/app.log file.

7. Logging in Different Environments

You can set different log levels for different profiles:

```
# application-dev.properties  
logging.level.root=DEBUG
```

```
# application-prod.properties  
logging.level.root=ERROR
```