1. What is Spring?

Spring is an open-source framework for Java that provides a comprehensive infrastructure for building enterprise applications. It provides inversion-of-control container (for dependency injection) and set of libraries (for web, data access, security, etc.), allowing developers to focus on business logic than boilerplate plumbing, i.e., building high-quality applications without worrying about low-level details.

It provides comprehensive programming and configuration model for modern Java applications. At its core, Spring implements IoC and DI – meaning the framework takes over the responsibility of constructing and managing your objects (beans) and their dependencies. This IoC container is what makes Spring powerful: you define how your components interact, and Spring injects the required components at runtime, instead of your code instantiating them directly.

The core container provides the IoC/DI functionality. On top of that, there are modules for web development (Spring MVC), data access (JDBC, ORM), security (Spring Security), aspect-oriented programming (Spring AOP), and many others. These can be used individually or together.

Spring is also the foundation of a broader Spring ecosystem (or Spring platform) which includes additional projects like Spring Boot, Spring Data, Spring Cloud, and more. The Spring Framework remains at the core of this platform, providing the base features that all other Spring projects build upon.

1. What is Spring Boot?

Spring Boot is an open-source project built on the Spring Framework that makes it faster and easier to create Spring-based applications. It achieves this by providing opinionated defaults and auto-configuration out of the box, so you can get a stand-alone, production-grade application running with minimal configuration. The key idea is that Spring Boot takes an “opinionated” view of the Spring Platform: it preselects reasonable default configurations and library versions so that developers don’t have to manually set up everything from scratch. In other words, Spring Boot extends the Spring Framework by making many decisions for you (convention-over-configuration). This helps eliminate a lot of boilerplate setup that traditional Spring applications require (like XML configurations or manual bean setup).

What Spring Boot provides:  
Starter Dependencies: Spring Boot offers started POMs (like spring-boot-starter-web, spring-boot-starter-data-jpa, etc.) that bundle related libraries and Spring modules into a single dependency.

Embedded Server: Spring Boot applications are usually run as stand-alone apps with an embedded servlet server (Tomcat, Jetty). You can just run the app jar and it starts the server internally, you don’t need to deploy a WAR to an external server.

No Code Generation or XML Required: Spring Boot favors Java-based configuration and annotations. You typically don’t need any XML configuration. Everything can be set up via annotations like @SpringbootApplication and properties files.

Spring Boot is not a new framework from scratch, but rather a toolkit that uses Spring Framework behind the scenes and make Spring easier to use. Spring Boot applications are still Spring applications at the core, but with a lot of setup done automatically (for instance, creating a default configuration, or running an embedded server with just a click).

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication // Marks this as a Spring Boot app (enabling auto-config and component scan)

public class DemoApplication {

public static void main(String[] args) {

SpringApplication.run(DemoApplication.class, args);

}

}

1. What is the relation between Spring Platform and Spring Boot?

The Spring platform refers to the entire Spring ecosystem or projects maintained under the Spring umbrella. This includes the Spring Framework itself (core container, MVC, etc.) as well as projects like Spring Boot, Spring Data, Spring Security, and so on. Spring Boot is one of these projects in the platform that is built on top of the Spring Framework to provide an opionated, easy way to create Spring applications. In essence, Spring Boot uses and depends on the Spring platform’s components; it’s not a replacement for Spring, but a toolkit that works within the Spring ecosystem.

One analogy: if the Spring Framework is a powerful engine, Spring Boot is like a friendly ignition and control system that lets you start and use that engine more easily. The engine (Spring Framework) does the heavy work, and Spring Boot just makes it simpler to get it running correctly.

1. What is the relation between Spring Platform and Spring framework?

The Spring Framework is the core of the Spring platform​. The term “Spring platform” refers to the entire ecosystem of Spring projects (Spring Framework plus all its related projects like Spring Boot, Spring Data, etc.). In other words, Spring Framework provides the foundational features (IoC container, Spring MVC, etc.) and the Spring platform encompasses Spring Framework along with all the other Spring technologies built around it.

Relationship: The Spring Framework is central to the Spring platform. Most other Spring projects are built on top of it or rely on it. For example, Spring Boot’s auto-configuration relies on Spring Framework’s features, Spring Data JPA builds on Spring Framework’s data and transaction support, Spring Security plugs into the Spring container and web framework, and so forth. This is why it’s said that “Spring framework is the core of Spring platform.

Another way to see it: the Spring platform = Spring Framework plus additional specialized projects.

1. What is Dependency Injection and how is it done in the Spring platform/framework?

Dependency Injection is a design pattern, in which instead of a class directly instantiating its dependencies (using new), it receives those dependencies from elsewhere (for example, passed in via a constructor or set via a setter method, or fields). The “injection” refers to the act of supplying the dependency to the dependent object. In Spring, DI is accomplished by the IoC container (like BeanFactory or typically an ApplicationContext): you declare relationships between beans (via configuration or annotations), and the Spring container instantiates the needed objects and injects them into each other at runtime​. This leads to loosely-coupled, easier-to-test code.

You, as a developer, typically configure which classes are beans and how they should be wired together. In older versions of Spring this could be done in XML files (by listing beans and their dependencies), and in modern Spring it’s usually done with annotations and/or Java configuration. Either way, the container takes care of creating instances of your classes (beans) and injecting any required dependencies into them.

1. What is Inversion of Control (Ioc) and how is it related to Spring?

Inversion of Control (IoC) is a principle where the control of certain responsibilities (like object creation and flow of control) is inverted from the application code to an external container or framework. In Spring, IoC means the Spring container is in charge of managing the lifecycle of objects (beans) and their inter-dependencies.

summarized by the phrase: “Don’t call us, we’ll call you.” This is known as the Hollywood Principle​. It captures the idea that, rather than your code calling into a library to get things done, you hand over control to a framework, and the framework calls your code at appropriate times. In traditional programming, your application is the boss: it decides when to create objects, when to invoke methods, etc. With IoC, you delegate some of these responsibilities to a container or framework.

You define beans and how they should be wired, and then at runtime Spring *inverts* the control by creating those objects and injecting dependencies, rather than you creating them manually. Your application classes just declare dependencies (for example, through constructor parameters or @Autowired fields) and do not worry about where those dependencies come from.

Spring also controls the execution flow in certain scenarios – for example, in a web application, Spring’s DispatcherServlet (part of Spring MVC) will control which controller method to call for a given request (so your controller methods are called by the framework, not by your own code directly).

When Spring’s container injects a dependency into a class, that’s IoC – the class isn’t controlling how it gets the dependency, the container is. From the class’s perspective, the control over its own dependencies has been inverted; it just gets handed what it needs. So, IoC and DI are closely related: DI is a pattern to implement IoC.