What is Spring boot:

Spring Boot is an open-source Java framework used to create a Micro Service. Spring boot is developed by Pivotal Team, and it provides a faster way to set up and an easier, configure, and run both simple and web-based applications. It is a combination of Spring Framework and Embedded Servers.

Why use spring boot framework?

- Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run" and reduces manual configurations.
- Spring Boot applications can include an embedded server, such as Tomcat or Jetty, within the application itself. This eliminates the need to deploy your application to a separate web server.
  Spring Boot provides pre-built packages called "starters" that include common dependencies for various functionalities like web development, data access, and security. This makes it easy to get started with specific features.
- Example: You want to connect to a database. Include the appropriate Spring Boot starter for your database (e.g., MySQL or PostgreSQL). Spring Boot will automatically configure the connection pool and basic interactions.

Pros:

- Faster Development: Less code, pre-built components for common tasks.
- Standalone Apps: Embedded server, ideal for microservices.
- Production-Ready: Built-in health checks and monitoring features.
- Large Community: Extensive resources and support available.

Cons:

- Learning Curve: Requires understanding core Spring concepts.
- Less Flexibility: Automatic configuration can be hard to override.
- Unused Dependencies: Might include features you don't need.
- Limited Control: Less fine-grained control over complex configurations.

Goal:
The main goal of the Spring Boot framework is to reduce overall development time and increase efficiency by having a default setup for unit and integration tests.

SPRING 1.x Module:

Spring 1.x was the first major release of the Spring Framework, and it introduced several core features that laid the foundation for what Spring has become today.

1. Inversion of Control (IoC) Container:

In Spring 1.x, the Inversion of Control (IoC) container was central to the framework. It manages Java objects (beans) and their dependencies.

Example XML configuration (beans.xml):
xmlCopy code<beans><bean id="userService" class="com.example.UserService"><property name="userRepository" ref="userRepository"/></bean><bean id="userRepository" class="com.example.UserRepository"/></beans>
Java code to use beans:

javaCopy codepublic class MyApp {
  public static void main(String[] args) {
  ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
  UserService userService = (UserService) context.getBean("userService");
  // Use userService...
  }
}

2. Aspect-Oriented Programming (AOP):

Spring 1.x introduced Aspect-Oriented Programming (AOP) capabilities, allowing developers to define cross-cutting concerns separately from the main application logic.

Example AOP configuration (aspect.xml):
xmlCopy code<beans><aop:config><aop:aspect ref="loggingAspect"><aop:pointcut id="businessService" expression="execution(* com.example.BusinessService.*(..))"/><aop:before pointcut-ref="businessService" method="logBefore"/></aop:aspect></aop:config><bean id="loggingAspect" class="com.example.LoggingAspect"/><bean id="businessService" class="com.example.BusinessService"/></beans>

Java code for the aspect:

javaCopy codepublic class LoggingAspect {
  public void logBefore(JoinPoint joinPoint) {
  System.out.println("Before executing method: " +
joinPoint.getSignature().getName());
  }
}

3. Dependency Injection (DI):

Dependency Injection is a core principle in Spring, enabling loose coupling between components.

Example bean definition with dependency injection:
xmlCopy code<bean id="userService" class="com.example.UserService"><property name="userRepository" ref="userRepository"/></bean><bean id="userRepository" class="com.example.UserRepository"/>
Java code in the UserService class:

javaCopy codepublic class UserService {
  private UserRepository userRepository;
 	
  public void setUserRepository(UserRepository userRepository) {
  this.userRepository = userRepository;
  }
}

4. Autoconfiguration: This feature eliminated a significant amount of manual configuration by automatically setting up beans based on the libraries you included in your project. Spring Boot detected these libraries and configured beans accordingly, saving you time and effort.

5. **Starters:** Starters were pre-configured dependency sets for common functionalities like web development, database access, security, and more. They simplified adding these features to your application and ensured compatibility between different libraries within a starter.

6. **Embedded Servers:** Spring Boot allowed you to package embedded servers like Tomcat, Jetty, or Undertow within your application JAR. This enabled you to run your application as a standalone executable without needing a separate server deployment.

7**. Actuator:** This module provided production-ready features for monitoring and managing your application. Actuator offered endpoints for:

Health Checks: Checking the overall health of your application and identifying potential issues.

Metrics: Gathering metrics about application performance, such as memory usage and thread count.

Configuration Information: Accessing your application's configuration details.

Basic Management: (Limited in 1.x) Performing basic management tasks like shutdown (use with caution due to security implications).

SPRING 2.x Module:

Spring Boot 2.x introduced several new features and improvements over Spring Boot 1.x, reflecting the evolution of the framework to better support modern application development practices. Here are some of the key enhancements and new features introduced in Spring Boot 2.x compared to 1.x:

1. **Java 8 and 9+ Support:**
   - Spring Boot 2.x requires Java 8 as a minimum and provides enhanced support for Java 9 and later versions, leveraging new language features and APIs.

2. **Default Embedded Server:**
   - Tomcat remains the default embedded server, but support for other embedded servers like Jetty and Undertow has been enhanced.

3. **Actuator Enhancements:**
   - Major overhaul of the Actuator module, with a redesigned set of endpoints and integration with Micrometer for better metrics collection and monitoring.
   - More customizable and flexible health checks and monitoring endpoints.

4. **Built on Spring Framework 5.x**

   Core Upgrade:

   - 2.x: Built on Spring Framework 5.x, offering better performance and modern programming support.
   - 1.x: Relied on Spring Framework 4.x, lacking newer features and enhancements.

   Reactive Foundation:

   - 2.x: Introduces support for reactive programming with Project Reactor.
   - 1.x: Did not have native support for reactive programming.

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

5. **Spring WebFlux Introduction**

   Non-blocking Architecture:

   - 2.x: Employs Spring WebFlux for efficient handling of concurrent connections.
   - 1.x: Focused on synchronous web applications using Spring MVC.

   Reactive Streams API:

   - 2.x: Utilizes Mono and Flux for asynchronous processing.
   - 1.x: Did not support the Reactive Streams API.

```
Example:
@RestController
public class ReactiveController {
  @GetMapping("/hello")
  public Mono<String> hello() {
    return Mono.just("Hello, Reactive World!");
  }
}
```

6. **Dual Programming Model Support**

Dual Support:

- 2.x: Allows both synchronous (Spring MVC) and asynchronous (Spring WebFlux) programming.
- 1.x: Limited to synchronous programming with Spring MVC.

Flexible Configuration:

- 2.x: Both models can coexist within the same application.
- 1.x: Restricted to synchronous configuration.

Example:

```
// Synchronous Controller
@RestController
@RequestMapping("/sync")
public class SyncController {
  @GetMapping("/hello")
  public String helloSync() {
    return "Hello, Synchronous World!";
  }
}

// Asynchronous Controller
@RestController
@RequestMapping("/async")
public class AsyncController {
  @GetMapping("/hello")
  public Mono<String> helloAsync() {
    return Mono.just("Hello, Asynchronous World!");
  }
}
```

SPRING 3.x Module:

- Java Baseline: Spring Boot 3 requires Java 17 as the minimum supported version. This means leveraging newer Java features like sealed classes and records (if applicable).
- Spring Framework and Spring Security: Spring Boot 3 relies on newer versions of these core frameworks (Spring Framework 6 and Spring Security 6). You'll need to adjust your project's dependencies accordingly.
- Enhanced Testing Support: Spring Boot 3 offers improvements for testing applications, including better integration with testing frameworks like JUnit 5. This can streamline your unit and integration testing processes.
- Spring Boot 3 removes the use of the org.springframework.boot.autoconfigure.EnableAutoConfiguration key in spring.factories for registering auto-configurations.
- Spring Expression Language (SpEL): A powerful expression language for querying and manipulating objects during runtime.

SPRING 4.x Module:

Bean Definition:

In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container.

Inversion of Control (IoC) is a process in which an object defines its dependencies without creating them. This object delegates the job of constructing such dependencies to an IoC container.