How Spring Boot actually defines and configures beans (i.e., *where do the dependencies come from?*).

In Spring Boot, there are **multiple ways to configure beans and their dependencies**, ranging from explicit Java code to completely automatic detection.

Let's go through them clearly 👇

⚙️ 1. Component Scanning (Annotation-Based Configuration)

This is the **most common and convenient** way in Spring Boot.

Spring automatically scans your classpath (starting from the package of your main application class) for components annotated with these stereotypes:

- `@Component`
- `@Service`
- `@Repository`
- `@Controller` or `@RestController`
- `@ControllerAdvice` or `@RestControllerAdvice`

Those classes are automatically registered as **beans** in the Spring application context.

Example:

```
@Service
public class PaymentService {
    public void pay() {
        System.out.println("Payment processed");
    }
}

@Component
```

```
public class OrderService {
    private final PaymentService paymentService;

    public OrderService(PaymentService paymentService) {
        this.paymentService = paymentService;
    }
}
```

Because both are annotated with `@Service` and `@Component`, Spring Boot automatically discovers and wires them.

✅ **Pros:**

- No XML needed
- Minimal configuration
- Automatically handled via classpath scanning

🧩 2. Java-Based Configuration (`@Configuration` + `@Bean`)

You can **explicitly define beans** in a Java configuration class using `@Bean`.

Example:

```
@Configuration
public class AppConfig {

    @Bean
    public PaymentService paymentService() {
        return new PaymentService();
    }

    @Bean
    public OrderService orderService(PaymentService paymentService) {
        return new OrderService(paymentService);
    }
}
```

Here, Spring Boot creates and manages these beans, injecting dependencies just as if they were annotated with `@Component`.

✅ **Pros:**

- Full control over bean creation
- Useful for integrating third-party libraries (where you can't annotate their classes)

🧰 3. XML Configuration (Legacy, Rarely Used in Boot)

Although Spring Boot favors Java and annotation-based configuration, **XML configuration** is still supported for legacy projects.

Example:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="paymentService" class="com.example.PaymentService" />
    <bean id="orderService" class="com.example.OrderService">
        <constructor-arg ref="paymentService"/>
    </bean>

</beans>
```

✅ **Pros:**

- Works for old Spring applications

⚠️ **Cons:**

- Verbose, less readable
- Rarely used in new Boot apps

⚡ 4. Programmatic Configuration (Using `ApplicationContext`)

You can register beans **manually** in code — not common, but possible.

Example:

```
@SpringBootApplication
public class MyApp {

    public static void main(String[] args) {
        var context = SpringApplication.run(MyApp.class, args);
        PaymentService paymentService =
context.getBean(PaymentService.class);
        paymentService.pay();
    }
}
```

Or even:

```
context.registerBean(MyCustomBean.class);
```

✅ **Pros:**

- Dynamic runtime configuration possible

⚠️ **Cons:**

- Rarely used; breaks declarative configuration style

✅ Summary Table

Configuration Method	**Description**	**Typical Use Case**
Component Scanning (`@Component`, `@Service`, etc.)	Auto-detects annotated beans	Most common
Java Config (`@Configuration + @Bean`)	Manually define beans	Third-party or custom initialization
XML Configuration	Old style	Legacy support
Programmatic (via Context)	Manual registration	Dynamic runtime logic