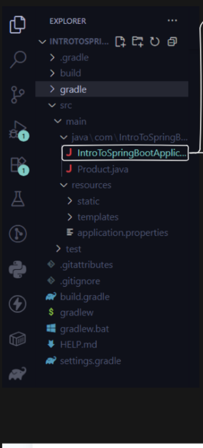
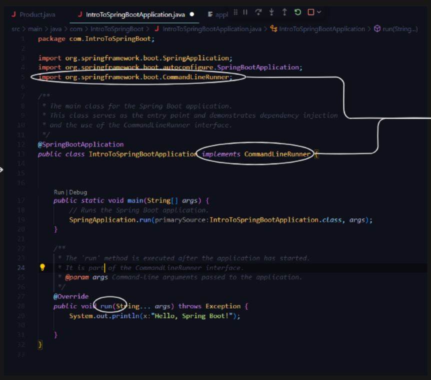
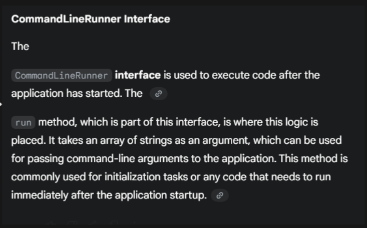
Intro To SPRINGBOOT Application:



This is main File from where Spring app RUNs !!!


Product.java
IntroToSpringBootApplication.java
appl
src > main > java > com > IntroToSpringBoot > IntroToSpringBootApplication.java > IntroToSpringBootApplication > run(String...)

```java
package com.IntroToSpringBoot;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.CommandLineRunner;

/**
 * The main class for the Spring Boot application.
 * This class serves as the entry point and demonstrates dependency injection
 * and the use of the CommandLineRunner interface.
 */
@SpringBootApplication
public class IntroToSpringBootApplication implements CommandLineRunner {



    Run | Debug
    public static void main(String[] args) {
        // Runs the Spring Boot application.
        SpringApplication.run(primarySource:IntroToSpringBootApplication.class, args);
    }

    /**
     * The 'run' method is executed after the application has started.
     * It is part of the CommandLineRunner interface.
     * @param args Command-line arguments passed to the application.
     */
    @Override
    public void run(String... args) throws Exception {
        System.out.println(x:"Hello, Spring Boot!");

    }
}
```

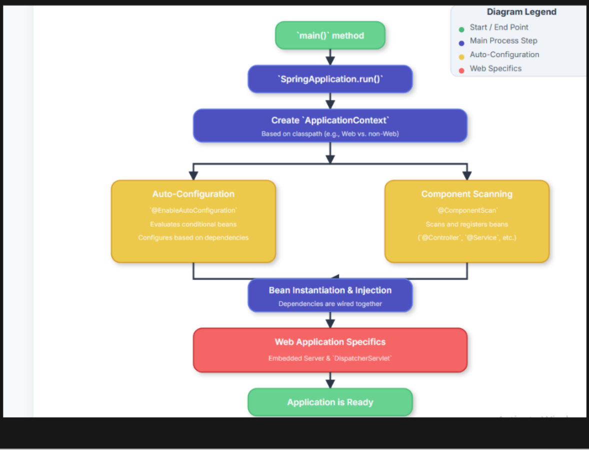


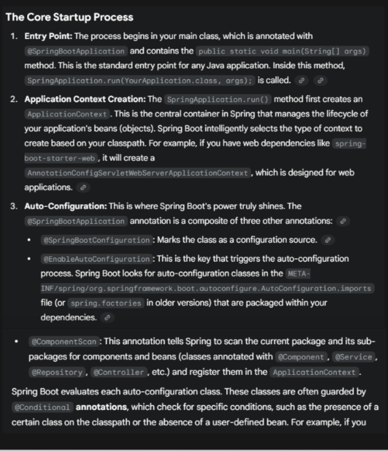
CommandLineRunner Interface

The

`CommandLineRunner` **interface** is used to execute code after the application has started. The

`run` method, which is part of this interface, is where this logic is placed. It takes an array of strings as an argument, which can be used for passing command-line arguments to the application. This method is commonly used for initialization tasks or any code that needs to run immediately after the application startup.


`main()` method
`SpringApplication.run()`
Create `ApplicationContext`
Based on classpath (e.g., Web vs. non-Web)
Auto-Configuration
`@EnableAutoConfiguration`
Evaluates conditional beans
Configures based on dependencies
Component Scanning
`@ComponentScan`
Scans and registers beans
(`@Controller`, `@Service`, etc.)
Bean Instantiation & Injection
Dependencies are wired together
Web Application Specifics
Embedded Server & `DispatcherServlet`
Application is Ready
Diagram Legend
● Start / End Point
● Main Process Step
● Auto-Configuration
● Web Specifics

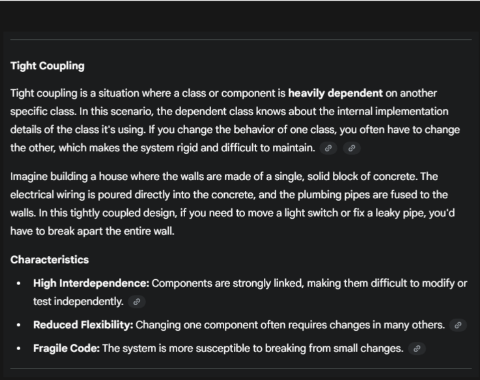


The Core Startup Process

1. **Entry Point:** The process begins in your main class, which is annotated with `@SpringBootApplication` and contains the `public static void main(String[] args)` method. This is the standard entry point for any Java application. Inside this method, `SpringApplication.run(YourApplication.class, args);` is called.
2. **Application Context Creation:** The `SpringApplication.run()` method first creates an `ApplicationContext`. This is the central container in Spring that manages the lifecycle of your application's beans (objects). Spring Boot intelligently selects the type of context to create based on your classpath. For example, if you have web dependencies like `spring-boot-starter-web`, it will create a `AnnotationConfigServletWebServerApplicationContext`, which is designed for web applications.
3. **Auto-Configuration:** This is where Spring Boot's power truly shines. The `@SpringBootApplication` annotation is a composite of three other annotations:
    - `@SpringBootConfiguration`: Marks the class as a configuration source.
    - `@EnableAutoConfiguration`: This is the key that triggers the auto-configuration process. Spring Boot looks for auto-configuration classes in the `META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports` file (or `spring.factories` in older versions) that are packaged within your dependencies.

- `@ComponentScan`: This annotation tells Spring to scan the current package and its sub-packages for components and beans (classes annotated with `@Component`, `@Service`, `@Repository`, `@Controller`, etc.) and register them in the `ApplicationContext`.

Spring Boot evaluates each auto-configuration class. These classes are often guarded by `@Conditional` **annotations**, which check for specific conditions, such as the presence of a certain class on the classpath or the absence of a user-defined bean. For example, if you



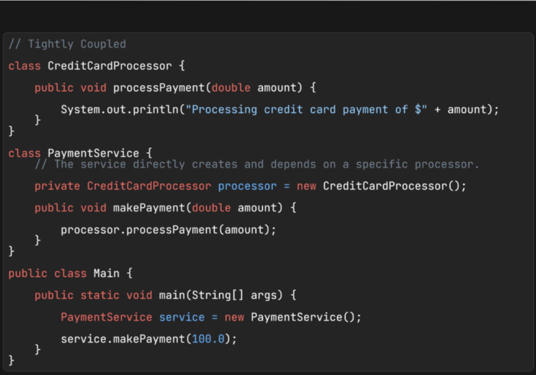
Tight Coupling

Tight coupling is a situation where a class or component is **heavily dependent** on another specific class. In this scenario, the dependent class knows about the internal implementation details of the class it's using. If you change the behavior of one class, you often have to change the other, which makes the system rigid and difficult to maintain.

Imagine building a house where the walls are made of a single, solid block of concrete. The electrical wiring is poured directly into the concrete, and the plumbing pipes are fused to the walls. In this tightly coupled design, if you need to move a light switch or fix a leaky pipe, you'd have to break apart the entire wall.

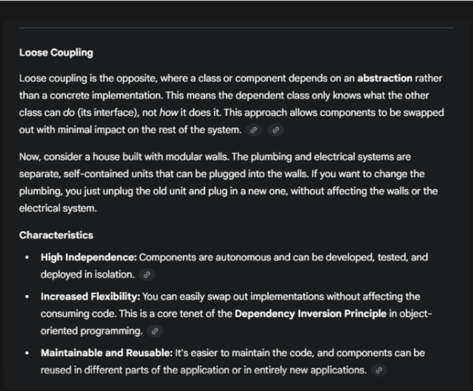
Characteristics

- **High Interdependence:** Components are strongly linked, making them difficult to modify or test independently.
- **Reduced Flexibility:** Changing one component often requires changes in many others.
- **Fragile Code:** The system is more susceptible to breaking from small changes.



```java
// Tightly Coupled
class CreditCardProcessor {
    public void processPayment(double amount) {
        System.out.println("Processing credit card payment of $" + amount);
    }
}
class PaymentService {
    // The service directly creates and depends on a specific processor.
    private CreditCardProcessor processor = new CreditCardProcessor();
    public void makePayment(double amount) {
        processor.processPayment(amount);
    }
}
public class Main {
    public static void main(String[] args) {
        PaymentService service = new PaymentService();
        service.makePayment(100.0);
    }
}
```

In this example, the `PaymentService` is **tightly coupled** to the `CreditCardProcessor`. It directly creates a new instance of the concrete `CreditCardProcessor` class.
If you wanted to use a different payment processor, say for PayPal, you would have to modify the `PaymentService` class directly. This makes the code rigid and hard to change.



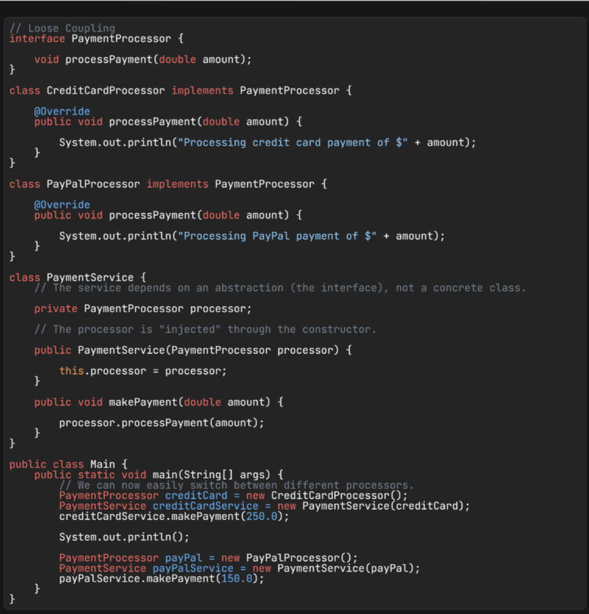
Loose Coupling

Loose coupling is the opposite, where a class or component depends on an **abstraction** rather than a concrete implementation. This means the dependent class only knows what the other class can *do* (its interface), not *how* it does it. This approach allows components to be swapped out with minimal impact on the rest of the system.

Now, consider a house built with modular walls. The plumbing and electrical systems are separate, self-contained units that can be plugged into the walls. If you want to change the plumbing, you just unplug the old unit and plug in a new one, without affecting the walls or the electrical system.

Characteristics

- **High Independence:** Components are autonomous and can be developed, tested, and deployed in isolation.
- **Increased Flexibility:** You can easily swap out implementations without affecting the consuming code. This is a core tenet of the **Dependency Inversion Principle** in object-oriented programming.
- **Maintainable and Reusable:** It's easier to maintain the code, and components can be reused in different parts of the application or in entirely new applications.



```java
// Loose Coupling
interface PaymentProcessor {
    void processPayment(double amount);
}
class CreditCardProcessor implements PaymentProcessor {
    @Override
    public void processPayment(double amount) {
        System.out.println("Processing credit card payment of $" + amount);
    }
}
class PayPalProcessor implements PaymentProcessor {
    @Override
    public void processPayment(double amount) {
        System.out.println("Processing PayPal payment of $" + amount);
    }
}
class PaymentService {
    // The service depends on an abstraction (the interface), not a concrete class.
    private PaymentProcessor processor;
    // The processor is "injected" through the constructor.
    public PaymentService(PaymentProcessor processor) {
        this.processor = processor;
    }
    public void makePayment(double amount) {
        processor.processPayment(amount);
    }
}
public class Main {
    public static void main(String[] args) {
        // We can now easily switch between different processors.
        PaymentProcessor creditCard = new CreditCardProcessor();
        PaymentService creditCardService = new PaymentService(creditCard);
        creditCardService.makePayment(250.0);

        System.out.println();

        PaymentProcessor payPal = new PayPalProcessor();
        PaymentService payPalService = new PaymentService(payPal);
        payPalService.makePayment(150.0);
    }
}
```

To achieve loose coupling, we use an **interface** to define a contract. The `PaymentService` now depends on the `PaymentProcessor` interface, not a concrete implementation. This is a core concept of the **Dependency Inversion Principle**.

Now, you can "inject" any class that implements the `PaymentProcessor` interface into the `PaymentService`. This makes the code flexible and easy to maintain.