**Use of @Profile**

The @Profile annotation in Spring Boot is used to conditionally enable or disable components based on which profile is currently active.

// This payment service will only be used in development/testing

@Service

@Profile("dev")

public class MockPaymentService implements PaymentService {

@Override

public void processPayment(Order order) {

// Just pretend to process payment

System.out.println("DEV: Simulating payment for order: " + order.getId());

}

}

// This is the real payment service that will be used in production

@Service

@Profile("prod")

public class StripePaymentService implements PaymentService {

@Override

public void processPayment(Order order) {

// Real Stripe API call to process actual payment

stripeClient.createCharge(order.getAmount(), order.getCustomerId());

}

}

When you run your application:

* If spring.profiles.active=dev, the MockPaymentService will be used
* If spring.profiles.active=prod, the StripePaymentService will be used

This is super useful when you want to:

* Use simplified/mock implementations during development
* Switch between different implementations based on environment
* Enable/disable certain features in different environments
* Have different configurations for different scenarios

The beauty is that your other code doesn't need to know which implementation it's using - Spring handles that based on the active profile!

**For Method Level:**

The @Profile annotation at method level can only be used with @Bean methods in @Configuration classes. Here's the correct way:  
@Configuration

public class PaymentConfig {

@Bean

@Profile("dev")

public PaymentService devPaymentService() {

return new PaymentService() {

@Override

public void processPayment(Order order) {

System.out.println("DEV: Simulating payment for order: " + order.getId());

}

};

}

@Bean

@Profile("prod")

public PaymentService prodPaymentService() {

return new PaymentService() {

@Override

public void processPayment(Order order) {

stripeClient.createCharge(order.getAmount(), order.getCustomerId());

}

};

}

}  
  
If you want to have different method implementations in a single service class, you might want to consider using conditional logic instead:

java

Copy

@Service

public class PaymentService {

@Value("${spring.profiles.active}")

private String activeProfile;

public void processPayment(Order order) {

if ("dev".equals(activeProfile)) {

*// Development implementation*

System.out.println("DEV: Simulating payment for order: " +order.getId());

}

else {

*// Production implementation*

stripeClient.createCharge(order.getAmount(), order.getCustomerId());

}

}

}

But generally, separating implementations into different classes (as in your original example) is considered better practice because it:

1. Follows the Single Responsibility Principle
2. Makes the code more maintainable
3. Is easier to test
4. Provides clearer separation of concerns

# If i am using @Profile at the method level , then at the Service Layer also I need to ensure that I use this annotation and write the business logic accordingly so that I can use a particular bean at the time.

No, that's not correct. When using @Profile with @Bean methods, you don't need to modify your service layer. Let me explain with an example:

java

Copy

*// First, define an interface*

public interface PaymentService {

void processPayment(Order order);

}

*// Configuration class with profile-specific beans*

@Configuration

public class PaymentConfig {

@Bean

@Profile("dev")

public PaymentService devPaymentService() {

return order -> {

System.out.println("DEV: Simulating payment for order: " + order.getId());

};

}

@Bean

@Profile("prod")

public PaymentService prodPaymentService() {

return order -> {

*// Real payment processing*

stripeClient.createCharge(order.getAmount(), order.getCustomerId());

};

}

}

//SEPARATE CLASS

*// Your service layer stays clean and profile-agnostic*

@Service

public class OrderService {

private final PaymentService paymentService; *// Spring injects the correct implementation*

public OrderService(PaymentService paymentService) {

this.paymentService = paymentService;

}

public void placeOrder(Order order) {

*// Business logic*

validateOrder(order);

*// The correct payment implementation is automatically injected*

*// based on the active profile*

paymentService.processPayment(order);

*// More business logic*

sendConfirmation(order);

}

}

Key points:

1. The service layer (OrderService) doesn't need to know about profiles
2. Spring automatically injects the correct PaymentService implementation based on the active profile
3. Your business logic remains clean and focused
4. The profile-specific behavior is entirely handled by Spring's dependency injection

This is one of the benefits of dependency injection - your business logic remains decoupled from the specific implementations.