

Class-34: Advanced Profile Management



by Pial Kanti Samadder

Activating Profiles (CLI)

Command Line:

```
java -jar app.jar --spring.profiles.active=dev
```

Using Maven:

```
mvn spring-boot:run -Dspring-boot.run.profiles=local
```

Activating Profiles (IDE)

IntelliJ Run Configurations

- Add VM option:

```
-Dspring.profiles.active=dev
```

- Or add environment variable:

```
SPRING_PROFILES_ACTIVE=dev
```

Environment-Specific Beans in Spring

Why We Need Conditional Bean Activation

- Applications run in multiple environments: **dev, test, prod**
- Each environment may require **different configurations**, e.g.
 - Different database connections
 - Different message brokers
 - Mock vs real services
- Hard-coding environment differences leads to:
 - Repeated code
 - Manual changes before deployment
 - Risk of misconfigurations
- Spring provides **@Profile** and **@Conditional** to solve this cleanly.

What Is @Profile?

@Profile allows you to activate beans **only when specific profiles are active**.

Key Idea

"Same codebase, different behavior depending on environment."

When to Use

- Dev-only beans (e.g., H2 DB, mock services)
- Prod-only beans (e.g., real payment service)
- Feature toggles (profile-based)
- Switching between real & stub integrations

Why @Profile Is Needed

Benefits

- Avoids hardcoding environment logic (if(dev) { ... })
- Keeps beans cleanly separated
- Eliminates code changes before deployment
- Ensures correct beans load automatically per environment
- Simplifies CI/CD pipelines

Typical Setup

```
spring.profiles.active=dev  
spring.profiles.active=prod
```

@Profile Example

Scenario

You want different database configs for development and production.

Dev Configuration

```
@Component  
@Profile("dev")  
public class DevDataSourceConfig implements DataSourceConfig {  
    @Override  
    public String url() {  
        return "jdbc:h2:mem:testdb";  
    }  
}
```

Prod Configuration/

```
@Component  
@Profile("prod")  
public class ProdDataSourceConfig implements DataSourceConfig {  
    @Override  
    public String url() {  
        return "jdbc:postgresql://prod-server/db";  
    }  
}
```

- Only the bean matching the active profile is loaded.

What Is @Conditional?

@Conditional activates beans based on **custom conditions**, not just profiles.

When @Profile Is NOT Enough

- Load a bean **only if a class exists** in the classpath
- Load a bean **only if a property is set**
- Load a bean **only if a feature flag is enabled**
- Load specific implementations based on **runtime evaluation**

High-Level Idea

"@Profile is environment-based; @Conditional is logic-based."

@Conditional Example

Condition: Load bean only if feature is enabled

```
@Configuration  
public class PaymentConfig {  
  
    @Bean  
    @ConditionalOnProperty(  
        name = "feature.payment.stripe.enabled",  
        havingValue = "true"  
    )  
    public PaymentService stripeService() {  
        return new StripePaymentService();  
    }  
}
```

Application Properties

```
feature.payment.stripe.enabled=true
```

@Profile vs @Conditional

Feature	@Profile	@Conditional
Based on	Environment profile	Any runtime logic
Example use	Dev vs Prod beans	OS detection, classpath, properties
Flexibility	Limited	Very flexible
Custom rules	✗ No	✓ Yes
Best for	Environment separation	Smart runtime decisions

When to Use Which

Use @Profile When:

- You want clear environment separation
- Different configs for dev, test, prod
- Multiple beans grouped under a named environment

Use @Conditional When:

- Decisions depend on runtime variables
- Feature flags
- Optional dependencies
- Hardware/OS/classpath detection

Secrets Management Strategies

Why Secrets Management Matters

- Avoid hardcoding sensitive information (passwords, API keys) in code.
- Prevent accidental leaks in version control or logs.
- Ensure secure and flexible configuration for different environments.

Industry-Standard Strategies:

- **Environment Variables**

```
spring.datasource.password=${DB_PASSWORD}
```

- Store secrets outside the codebase.
- Easily override per environment (dev/test/prod).
- Works well with containers and cloud deployments.

- **External Secret Management Tools** (*Recommended for Production*)

- Examples: **HashiCorp Vault, AWS Secrets Manager, Azure Key Vault**.
- Centralized, dynamic, and audit-friendly secret storage.
- Supports automatic rotation and access control.

- **External Configuration Files (Non-Sensitive Config)**

- Use `application.properties` or `application.yml` for general configuration.
- Avoid storing secrets here; keep only environment-independent configs.

Best Practices:

- Never commit secrets to version control.
- Limit access based on roles and services.
- Rotate secrets regularly.
- Combine with **Spring Profiles** for environment-specific configuration.