Spring Tip #7

Use @Conditional Annotations to Enable/Disable Beans Dynamically

LAHIRU
LIYANAPATHIRANA

Context

Spring Framework provides powerful conditional configuration capabilities through its **@Conditional** annotations.

These annotations allow developers to enable or disable beans dynamically based on various conditions.

This feature enhances the flexibility of the application and promotes cleaner, maintainable, and configurable code.

Why Use Conditional Annotation?

@Conditional annotations enable dynamic bean registration. This feature provides several benefits:

- Enables feature toggling without modifying code.
- Allows different configurations for different environments.
- Loads beans when specific beans/classes are available.
- Provides fallback implementations when certain beans/classes are missing.

Common @Conditional Annotations

- **@ConditionalOnProperty**
- **@ConditionalOnBean**
- **@ConditionalOnMissingBean**
- **@ConditionalOnClass**
- **@ConditionalOnMissingClass**
- **@ConditionalOnExpression**

These are the most common annotations in Spring. Refer to the official Spring boot documentation for the other **@Conditional** annotations.

LAHIRU
LIYANAPATHIRANA

@ConditionalOnProperty

Enable a bean only if a specific property in application.properties (or application.yml) has a specified value.

```
@Bean
@ConditionalOnProperty(name = "app.feature.enabled", havingValue = "true")
public FeatureService featureService() {
  return new FeatureService();
}
```

Key Parameters:

- name: Property key (e.g., app.feature.enabled).
- havingValue: Required value (default: "true").
- matchIfMissing: If true, the bean is enabled even if the property is missing.

@ConditionalOnProperty

Use Cases:

- Feature Toggles: Enable/disable features (e.g., experimental UI, logging) via properties.
- Environment-Specific Configs: Load production-only beans (e.g., payment gateways) in prod environments.
- A/B Testing: Serve different implementations to user groups based on feature flags.

@ConditionalOnBean

Enable a bean only if a specific bean already exists in the application context.

```java
@Bean
@ConditionalOnBean(DataSource.class)
public JdbcTemplate jdbcTemplate(DataSource dataSource) {
  // Created only if DataSource is available
  return new JdbcTemplate(dataSource);
}
```

Key Parameters:

- value: The class of the bean to check (e.g., DataSource.class).
- name: The name of the bean (e.g., "dataSource").

Note: Bean creation order matters! Ensure the checked bean is defined earlier.

@ConditionalOnBean

Use Cases:

- Dependent Beans: Create a JdbcTemplate only if a DataSource bean is available.
- Plugin Systems: Enable a security filter only if an authentication service bean exists.
- Order-Sensitive Initialization: Ensure a bean (e.g., SchedulerService) loads after its dependencies.

@ConditionalOnMissingBean

Register a bean only if no other bean of the same type exists in the application context.

```
@Bean
@ConditionalOnMissingBean
public PaymentService paymentService() {
  // Fallback implementation
  return new DefaultPaymentService();
}
```

Use Cases:

- Default Implementations: Provide a basic CacheManager if no custom implementation exists.
- Avoid Duplicates: Prevent bean conflicts (e.g., skip a default DataSource if another is already defined).

@ConditionalOnClass

Enable a bean only if a specific class is present on the classpath.

```java
@Bean
@ConditionalOnClass(name = "org.postgresql.Driver")
public DatabaseService postgresService() {
  // Runs if PostgreSQL driver is available
  return new PostgresService();
}
```

Use Cases:

- Optional Integrations: Auto-configure integrations with optional libraries if the driver are in the classpath.
- Third-Party SDKs: Enable cloud storage (e.g., AWS S3) only if the SDK is included.

@ConditionalOnClass

Use Cases:

- Framework Compatibility: Load compatibility layers for legacy libraries if detected.

@ConditionalOnMissingClass

Enable a bean only if a specific class is missing from the classpath.

```java
@Bean
@ConditionalOnMissingClass("com.aws.s3.AmazonS3")
public StorageService storageService() {
  // Fallback if AWS SDK is missing
  return new LocalStorageService();
}
```

Use Cases:

- Fallback Implementations: Use local file storage if cloud SDKs are missing.
- Lightweight Modes: Provide basic implementations if optional libraries are excluded.

@ConditionalOnMissingClass

Use Cases:

- Legacy Support: Switch to older APIs when newer dependencies are unavailable.

@ConditionalOnExpression

Use SpEL (Spring Expression Language) to define complex conditions involving multiple properties or logic.

```java
@Bean
@ConditionalOnExpression("${app.cache.enabled} && '${app.env}' == 'prod'")
// CacheManager is enabled when app.cache.enabled is true
// and when app.env is prod
public CacheManager cacheManager() {
  return new RedisCacheManager();
}
```

Use Cases:

- Multi-Property Logic: Enable beans only when multiple conditions are met. (e.g., enable caching only if app.cache.enabled=true and app.env=prod)

@ConditionalOnExpression

Use Cases:

- Time-Based Rules: Activate a service during specific hours/days using SpEL date checks.
- Combining multiple flags.

Custom Conditions with @Conditional

Create custom logic by implementing the Condition interface. This can be used to create complex custom logic depending on the scenario.

```
public class CustomCondition implements Condition {
  @Override
  public boolean matches(ConditionContext context,
      AnnotatedTypeMetadata metadata) {

    return context.getEnvironment()
      .getProperty("app.custom.flag", Boolean.class, false);
  }
}

@Bean
@Conditional(CustomCondition.class)
public CustomBean customBean() {
  // CustomBean is enabled when app.custom.flag is true
  return new CustomBean();
}
```

Summary

Using **@Conditional** annotations allows dynamic bean registration, improving flexibility, maintainability, and feature toggling in Spring applications.

Finally:

✅ Use **@ConditionalOnProperty** for property-based conditions.

✅ Use **@ConditionalOnMissingBean** for default implementations.

✅ Use **@ConditionalOnBean** for dependent beans.

LAHIRU
LIYANAPATHIRANA

Summary

✅ Use **@ConditionalOnClass** or **@ConditionalOnMissingClass** for classpath-based conditions.

✅ Use **@ConditionalOnExpression** for SpEL-based conditions.

✅ Create custom conditions with **@Conditional**.

LAHIRU
LIYANAPATHIRANA

Did You Find This Post Useful?

Stay Tuned for More Spring Related Posts Like This

LAHIRU
LIYANAPATHIRANA