Technical Specification

Introduction to Spring's Dependency Injection

Introduction

This technical specification explains the mechanism of dependency injection and the conventions for applying it in Spring Boot. It is designed to be self-contained, providing sufficient information for comprehensive understanding.

If you are unfamiliar with the IoC Container or Dependency Injection, refer to *A Brief Introduction to Spring Boot*, which covers the essential concepts needed to understand this specification.

SpringBoot’s Stereotype

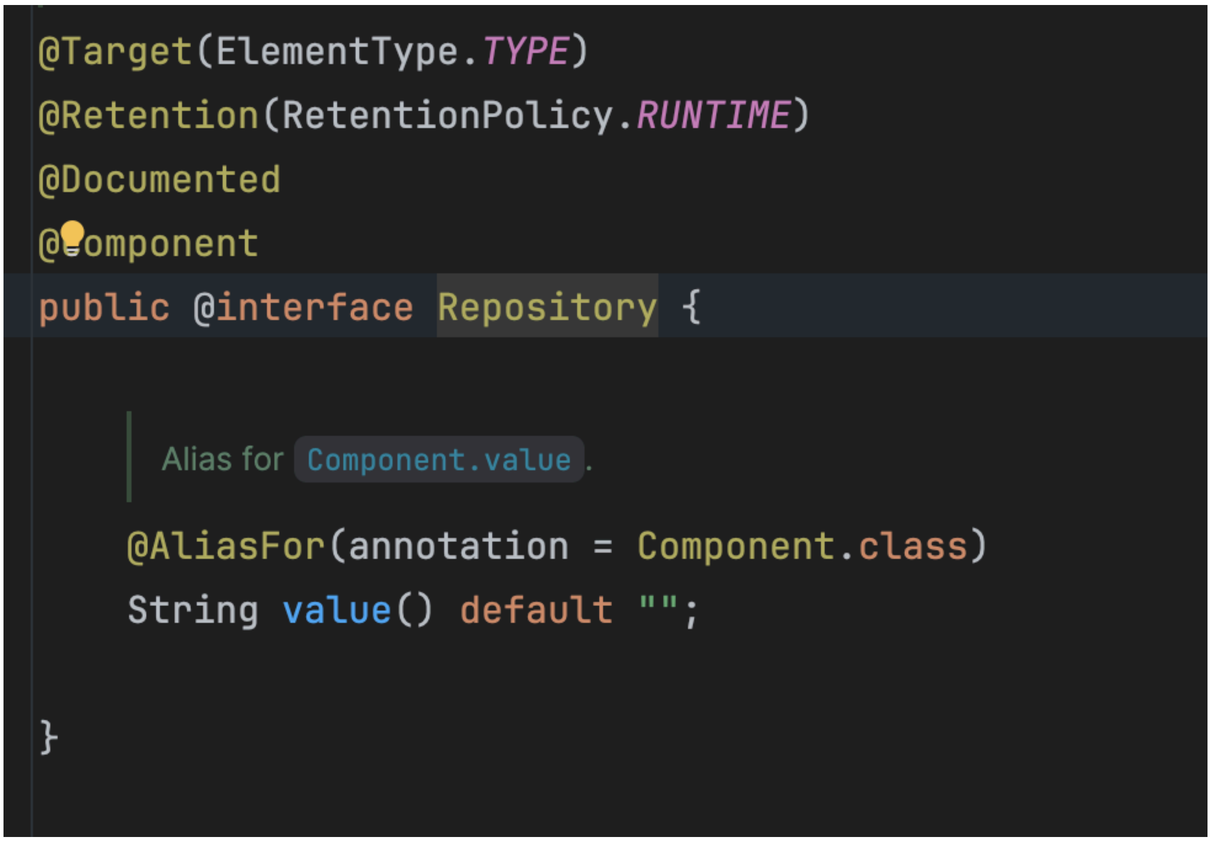
What Are Stereotypes in Spring Boot?

Spring Boot provides four commonly used stereotypes: @Component, @Service, @Repository, and @RestController (we will skip @Controller).

All of them serve the same fundamental purpose: ***to register a class as a bean in the IoC container***. Once registered, all required dependencies of the class are automatically injected, making the class available for injection and use by other components.

Internally, all these annotations are specializations of @Component. As stated in the Javadoc: *This annotation serves as a specialization of @Component, allowing for implementation classes to be autodetected through classpath scanning.*

This behavior is also confirmed in the source code.



```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Repository {

    Alias for Component.value.
    @AliasFor(annotation = Component.class)
    String value() default "";

}
```

Defining Your Own Stereotype

You can define your own stereotype annotation. The format is simple — replace `YourStereotypeName` with your desired name:

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface YourStereotypeName {

   /**
    * Alias for {@link Component#value}.
    */
   @AliasFor(annotation = Component.class)
   String value() default "";

}
```

When Should You Define a Custom Stereotype?

Before creating a custom stereotype, ensure it adds clear semantic or functional value. A new stereotype should only be introduced when one of the following conditions is met:

1. To **convey semantic meaning.**
   a. For example, defining @UseCase to represent domain-specific use cases in Domain-Driven Design (DDD).
   b. Semantic annotations improve readability and clarify the role or layer of a class.
2. To **enable Aspect-Oriented Programming (AOP).**
   a. For instance, creating annotations for cross-cutting concerns such as logging, validation, or auditing.
   b. Specialized annotations can help standardize AOP behavior and reduce repetitive code.

Otherwise, prefer the standard stereotypes, i.e., @Service, @Repository, and @RestController, as they are widely recognized and enhance code readability. Introducing custom stereotypes without clear documentation or intent increases cognitive load and may confuse future developers.

@Configuration and @Bean

There are two primary ways to create a bean in Spring:

1. By annotating a class with one of Spring's stereotypes (as discussed earlier).
2. By using @Configuration and @Bean annotations.

How to use @Configuration and @Bean?

To register beans using this method:

1. Annotate the class with @Configuration.
2. Annotate each factory method with @Bean.

Additional Rules and Considerations

1. The method should be public and non-final.
   a. While a private or final method annotated with @Bean can still register a bean, proxying will not work because CGLIB can only intercept public methods.
2. The class itself should be non-final.
   a. A final class annotated with @Configuration cannot be subclassed by Spring's CGLIB proxy, causing it to lose its interception benefits and potentially behave abnormally.

```
@Configuration
public class AppConfig {
    @Bean
    private MyService myService() {
        return new MyService();
    }

    @Bean
    public MyController controller() {
        // This will NOT go through the proxy; it will call th
private method directly
        return new MyController(myService());
        }
}
```

Spring uses CGLIB subclassing to intercept @Bean method calls within a @Configuration class. Since final methods cannot be overridden, Spring cannot insert interception logic, resulting in direct method calls without caching.
This leads to multiple bean instances being created instead of a singleton.

When to Use @Configuration and @Bean Instead of Stereotype?

In most cases, business or application logic should use stereotypes like @Service, @Repository, or @RestController.

Use @Configuration and @Bean primarily when:

1. **Integrating external libraries or frameworks**
    a. When integrating third-party or ***optional*** modules that provide utilities but are not automatically registered as beans.
    b. Examples include defining a reusable ObjectMapper, or customizing a component such as an AuthenticationProvider.
2. **For modular or reusable configurations**
    a. When creating generic or reusable packages that can be imported across multiple applications or modules.

When You Can Skip @Configuration

Sometimes, we intentionally omit the @Configuration annotation to make a set of beans optionally importable. In such cases, the beans can still be initialized using other approaches:

Other approaches include:

1. Imported manually (@Import(SimpleConfig.class)),

2. Declared in another configuration’s @Import,
3. Programmatic Registration (new AnnotationConfigApplicationContext(SimpleConfig.class)),

Note:

- @Configuration itself is a specialization of @Component.
- If the class is not registered (via @ComponentScan, @Import, or programmatically), its @Bean methods will not be detected or instantiated.
- Spring does not classpath-scan for @Bean methods directly; it only scans for stereotypes such as @Configuration, @Component, @Service, or @Controller.

SpringBoot Dependency Injection Mechanism

Spring Boot (and Spring Framework) follows a defined order of resolution when performing dependency injection. The process determines which bean should be injected when multiple candidates exist.

1. **Qualifier-Based Injection**
   1. If a dependency is annotated with @Qualifier, Spring Boot will use the specified qualifier to locate the matching bean.
   2. If multiple beans share the same name or qualifier, Spring will throw an error during the initialization phase.
2. **Primary Bean Reference**
   1. If more than one bean of the same type exists and one of them is annotated with @Primary, that bean is selected by default.
3. **Type-Based Injection (Default Behaviour)**
   1. If no qualifier is provided, Spring Boot first looks for a bean by type.
   2. For example, if a dependency of type LoggingFilter is required and only one bean of that type exists, Spring injects it automatically.
   3. If multiple beans of the same type are found, Spring will look for further hints (@Primary, @Qualifier, or bean name) to decide which one to inject.
4. **Name-Based Injection**
   1. If no @Primary or @Qualifier is present and multiple beans share the same type, Spring Boot will next attempt to match by bean name.
   2. The bean name can be derived from:
      1. The method name (for beans defined via @Bean), or
      2. The component name (for classes annotated with stereotypes like @Component or @Service).
5. **Error on Ambiguity**

1. If Spring Boot still cannot determine a unique bean after applying the above rules, it throws a NoUniqueBeanDefinitionException, indicating that the dependency cannot be resolved.

Key Takeaway:
Spring’s dependency injection prioritizes explicit qualifiers (@Qualifier or @Primary) before falling back to type and name-based matching. Always define qualifiers or primaries when multiple beans of the same type exist to avoid ambiguity.

Conventions

The following are conventions — opinionated but recommended practices for using @Configuration and @Bean. While not mandatory, following them improves readability, maintainability, and consistency across the codebase. A standardized approach helps ensure code quality and makes it easier for others to understand your configuration logic.

Convention 1: Use Method Name as Bean Name (Instead of Configured Name)

As discussed earlier, Spring Boot automatically uses the method name as the bean name. Therefore, prefer using the method name rather than explicitly specifying a name via @Bean and referencing it through @Qualifier. This reduces boilerplate and keeps the code clean.

In some cases (e.g., with @Async), you may still need to specify a bean name — for example, to identify a particular thread pool. However, in general, rely on the method name to minimize noise.

Drawback:
Changing a method name can break dependent beans.

Mitigation:

1. Enforce standardized, descriptive naming conventions.
2. Use compile-time and runtime validations to catch misconfigurations early.

Convention 2: Avoid Magic Strings for Bean Name

When you must specify a bean name, avoid using string literals like:

```
@Bean(value = "beanName")
@Bean(name = "beanName")
```

Instead, define and reference a constant:

```
@Bean(BEAN_NAME)
```

Reasons: Constants make renaming safer and more manageable during refactoring. The code is clearer and less error-prone, especially when multiple components depend on the same bean.

Convention 3: Avoid Class-Level Dependency Injection in @Configuration Classes

In older codebases, you may encounter class-level dependency injection, such as:

```
@RequiredArgsConstructor
@Configuration
public class Config {

    private final Dependency1 dependency1;

    private final Dependency2 dependency2;

    private final Dependency3 dependency3;

    @Bean
    public Service service() {
        return new Service(dependency1, dependency2, dependency3);
    }
}
```

This style can cause several issues:

1. It becomes unclear which dependencies belong to which beans.
2. Shared dependencies across beans can lead to coupling and confusion.

Recommended Approach:

```
@Configuration
public class Config {

    @Bean
    public Service service(Dependency1 dependency1,
                           Dependency2 dependency2,
                           Dependency3 dependency3) {
        return new Service(dependency1, dependency2,
dependency3);
    }
}
```

Each bean’s dependencies are now explicit and self-contained, making the code easier to understand without scrolling between fields and methods.

Convention 4: One Top-Level Dependent per @Configuration class

Each @Configuration class should define one top-level bean (the main dependent), with all supporting beans defined elsewhere. This keeps configurations logically grouped and self-contained.

Bad Example:

```
@Bean
public WebClient webClientA(WebClient.Builder builder) {
    return builder.build();
}

@Bean
public WebClient webClientB(WebClient.Builder builder,
                            Filter filterA) {
```

```
    return builder
            .filter(filterA)
            .build();
}
```

Here, both webClientA and webClientB are top-level beans defined in the same configuration file. If filterA is also defined here, it mixes unrelated concerns; if it's defined elsewhere, the configuration becomes fragmented.

Recommended Approach:

```
@Configuration
public class WebClientAConfig {
    @Bean
    public WebClient webClientA(WebClient.Builder builder) {
        return builder.build();
    }
}

@Configuration
public class WebClientBConfig {
    @Bean
    public WebClient webClientB(WebClient.Builder builder,
                                Filter filterA) {
        return builder.build();
    }
}
```

This structure ensures:

1. Each configuration file contains one logical unit.
2. Dependencies are clearly scoped and self-contained.
3. Shared beans (like Filter) can be placed in separate configuration classes for reuse.

Convention 5: Composition over Inheritance

In some legacy configurations, you may encounter inheritance-based setups such as:

```
@Configuration
public class ChildConfig extends ParentConfig {
    // Omitted for brevity
}
```

This approach is often used to reuse bean methods defined in the parent configuration. For example, ParentConfig might define a base dependency like baseBuilder(), which ChildConfig reuses for further customization. However, composition is preferred over inheritance, as it is more flexible, modular, and easier to maintain. Even languages like Go have eliminated inheritance entirely in favor of composition.

Recommended Approach:

```
@Configuration
public class ParentConfig {
    @Bean
    public Builder baseBuilder() {
        return new Builder();
    }
}

@Configuration
public class ChildConfig {
    @Bean
    public Dependency childDependency(Builder baseBuilder) {
        return builder.build();
    }
}
```

By exposing the base builder as a bean, other configuration classes can use it via dependency injection instead of inheritance. This promotes better separation of concerns and simplifies maintenance.

Convention 6: Dependency Injection over

Method Calls

In some configurations, developers directly call other @Bean methods to reuse their logic. While functional, this introduces method-level dependencies, adding unnecessary complexity beyond class-level and parameter-level dependencies. To maintain clarity and consistency, always prefer method parameter injection instead of method calls.

Bad Example:

```
@Bean
private WebClient.Builder
webClientBuilderA(ReactorClientHttpConnector
reactorClientHttpConnector) {
    return getBaseWebClientBuilder()
            .clientConnector(reactorClientHttpConnector)
            .observationRegistry(observationRegistry);
}
```

Recommended Approach:

```
@Bean
private WebClient.Builder webClientBuilderA(
    WebClient.Builder baseWebClientBuilder,
    ReactorClientHttpConnector reactorClientHttpConnector,
    ObservationRegistry observationRegistry
) {
    return baseWebClientBuilder
            .clientConnector(reactorClientHttpConnector)
            .observationRegistry(observationRegistry);
}
```

Using dependency injection through method parameters makes all dependencies explicit and easier to trace, ensuring consistency and testability.

Convention 7: Group Beans Without Special

Dependencies in a Single Configuration

This convention complements Convention 7.

In cases where multiple beans have similar structure and dependency patterns (e.g., defining multiple thread pools), creating a separate configuration class for each may be excessive.

In such scenarios, defining them in a single configuration class is acceptable — provided the grouping is logical and intentional.

Use this convention sparingly, only when:

1. The beans are of the same type and purpose.
2. Their dependencies are uniform.
3. Their only distinction lies in their names or configuration values.

This approach balances maintainability and simplicity without unnecessary fragmentation.

Convention 8: Naming Convention

To ensure clarity and consistency across the codebase, follow these naming standards:

1. Use nouns, not verbs, for bean and method names.
    a. ❌ getBaseBuilder()
    b. ✅ baseBuilder()
2. All configuration files must end with Config.
    a. Example: WebClientConfig, SecurityConfig, ObjectMapperConfig
3. Differentiate beans clearly using the <ClassName><Type> format.
    a. Configuration class: iFastPayWebClientConfig
    b. Bean method: iFastPayWebClientBuilder()

Consistent naming improves discoverability, refactoring safety, and team-wide readability.

Dependency Injection

Spring Boot supports three types of dependency injection:

1. **Constructor Injection**
   - Achieved using @Autowired, @Inject, or Lombok’s @RequiredArgsConstructor with final fields.
   - **Preferred approach** — promotes immutability, simplifies testing, and prevents runtime reassignments.
2. **Field Injection**
   - Achieved by annotating fields with @Autowired, @Inject, @Resource, or @Value.
   - Fields **cannot be final**.
   - Less favored due to testing and immutability limitations.
3. **Setter Injection**
   - Achieved by annotating setter methods with @Autowired, @Inject, @Resource, or @Value.
   - Fields **cannot be final**.
   - Useful for optional dependencies or when late initialization is necessary.

Best Practice:
Always favor **constructor injection** using private final fields and @RequiredArgsConstructor.
It improves testability, promotes immutability, and minimizes the risk of unintended runtime modifications.