# **A Short Note**

# ***Introduction to @Configuration and @Bean***

## **Introduction**

This note covers the mechanism and mainly the convention of using @Configuration and @Bean, which are the basis of programmatic configuration. It is designed to be self-contained, providing sufficient information for comprehensive understanding.

**Change History**

|  |  |  |
| --- | --- | --- |
| **Date** | **Version** | **Changelog** |
| 9 October 2025 | 1.0.0 | Initial docs. |

## 

## 

## 

## **Mechanism**

### **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.
   1. 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.
   1. 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 the 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**
   1. When integrating third-party or ***optional*** modules that provide utilities but are not automatically registered as beans.
   2. Examples include defining a reusable ObjectMapper, or customizing a component such as an AuthenticationProvider.
2. **For modular or reusable configurations**
   1. 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.

## 

## 

## **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.

### **1: [Recommended] Prefer Type-based/ @Qualifier Dependency Injection.**

To ensure safe refactoring and adhere to the *Principle of Least Astonishment*, prefer type-based dependency injection as the default approach. This enhances readability and predictability for maintainers.

**Bad Example**

|  |
| --- |
| public OncePerRequestFilter oncePerRequestFilter() {  return new OncePerRequestFilter() {  @Override  protected void doFilterInternal(  @NonNull HttpServletRequest request,  @NonNull HttpServletResponse response,  @NonNull FilterChain filterChain  ) throws ServletException, IOException {  // Your logic  }  };  } |

**Good Example**

|  |
| --- |
| @Component  public class CustomFilter extends OncePerRequestFilter {  @Override  protected void doFilterInternal(  @NonNull HttpServletRequest request,  @NonNull HttpServletResponse response,  @NonNull FilterChain filterChain  ) throws ServletException, IOException {  // Your logic  }  } |

However, this approach can lead to boilerplate code, especially when dealing with functional interface–like components such as GatewayFilter. In such cases, defining beans using @Configuration and @Bean with @Qualifier and lambda expressions can be more concise.

|  |
| --- |
| @Bean(BEAN\_NAME)  public GatewayFilter gatewayFilter() {  return (serverWebExchange, gatewayFilterChain) -> {  // Your logic here  };  }  @Bean  public SecurityWebFilterChain(@Qualifier(BEAN\_NAME) GatewayFilter gatewayFilter) {  // …  } |

However, explicitly specifying bean names with @Bean and @Qualifier can introduce unnecessary noise. To reduce verbosity, it is acceptable to use the method name as the bean name, following the convention over configuration principle.

|  |
| --- |
| @Bean  public GatewayFilter specialGatewayFilter() {  return (serverWebExchange, gatewayFilterChain) -> {  // Your logic here  };  }  @Bean  public SecurityWebFilterChain(GatewayFilter specialGatewayFilter) {  // …  } |

While this approach is concise, it can sometimes violate the Principle of Least Astonishment, as the dependency is now implicit and harder to refactor safely.

To balance clarity and convenience, follow this convention:

1. **Consistency Rule**: The method name, @Bean name, and @Qualifier should always be identical.
2. **Internal Beans**: If a bean is used only within the same @Configuration class, rely on the method name as the bean name.
3. **External Beans**: If a bean is used by beans in other configuration classes/ components, explicitly specify a @Qualifier.

Example:

|  |
| --- |
| @Configuration  public class AuthWebClientConfig {  public static final String AUTH\_WEB\_CLIENT = "authWebClient";  /\*  \* Referenced by beans outside this configuration class — specify a bean name.  \*/  @Bean(AUTH\_WEB\_CLIENT)  public WebClient authWebClient(  WebClient.Builder defaultWebClientBuilder,  Environment environment,  ExchangeFilterFunction authWebClientRemoveHeaderFilter) {  return defaultWebClientBuilder  .baseUrl(Objects.requireNonNull(environment.getProperty("iam.services.base.url")))  .filter(authWebClientRemoveHeaderFilter)  .build();  }  /\*  \* Used only within this configuration class — method name serves as the bean name.  \*/  @Bean  public ExchangeFilterFunction authWebClientRemoveHeaderFilter() {  return ExchangeFilterFunction.ofRequestProcessor(Mono::just);  }  } |

### **2: [Mandatory] 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.

**3: [Mandatory] 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.

**4: [Mandatory] @Configuration Splitting Rules**

These rules define when and how to split a Spring @Configuration class to maintain a clean, modular, and logically consistent structure.

Before diving into splitting rules, we must understand three key concepts: *Application Beans*, *Infrastructure Beans*, and *Dependents*.

* *Application beans* are aware of business logic or application-level concerns. They often represent workflows, policies, or NFRs (Non-Functional Requirements) that depend on domain logic.
  + E.g. SecurityFilterChain, WebClient, etc.
  + If the bean’s behavior changes when business rules change, it is an Application Bean.
* *Infrastructure* beans are business-agnostic and exist purely to support the system. They define reusable, low-level configurations (thread pools, encoders, mappers, etc.) used by multiple modules.
  + E.g. ThreadPoolTaskExecutor, PasswordEncoder, ObjectMapper
  + If the bean only defines system-level parameters (like thread pool size or timeout) and can be reused everywhere, it is an Infrastructure Bean.

All beans have some dependency relationships. To organize them, classify them into tiers based on their role and dependency depth.

A (Tier-1: Top-Level Dependent)

├─ B (Tier-2: Direct Support Bean)

│  ├─ D (Tier-3: Helper Bean)

│  └─ E (Tier-3: Helper Bean)

└─ C (Tier-2: Direct Support Bean)

   └─ F (Tier-3: Helper Bean)

Keep Tier-1, Tier-2, and Tier-3 beans that belong to the same logical feature within the same @Configuration file. If a new top-level bean (G) with its own dependencies appears, it should move to a new configuration. If multiple top-level beans (A and G) share Tier-2 beans (B and C), then extract B and C into a shared configuration.

**Rule 4.1: Single Top-Level Dependant Business Beans**

A top-level dependent is the primary bean that represents the entry point or integration boundary of a module. Each @Configuration file should define only one top-level dependent. All other beans in that file (application or infrastructure) should serve as dependencies for that top-level bean.

In the following example, the authWebClient is the top-level dependant, where the authWebClientRemoveHeaderFilter is its supporting dependency.

|  |
| --- |
| @Configuration  public class AuthWebClientConfig {  public static final String AUTH\_WEB\_CLIENT = "authWebClient";  /\*  \* Referenced by beans outside this configuration class — specify a bean name.  \*/  @Bean(AUTH\_WEB\_CLIENT)  public WebClient authWebClient(  WebClient.Builder defaultWebClientBuilder,  Environment environment,  ExchangeFilterFunction authWebClientRemoveHeaderFilter) {  return defaultWebClientBuilder  .baseUrl(Objects.requireNonNull(environment.getProperty("iam.services.base.url")))  .filter(authWebClientRemoveHeaderFilter)  .build();  }  /\*  \* Used only within this configuration class — method name serves as the bean name.  \*/  @Bean  public ExchangeFilterFunction authWebClientRemoveHeaderFilter() {  return ExchangeFilterFunction.ofRequestProcessor(Mono::just);  }  } |

**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();  }  } |

**Rule 4.2: Common Bean**

A common bean is any bean (application or infrastructure) used by two or more top-level dependents.

If multiple configurations define the same @Bean, extract it into:

1. A shared infrastructure configuration, or
2. A standalone @Component/ Util class, if it’s application-aware.

Bad Example

|  |
| --- |
| @Configuration  public class ConfigA {  @Bean  public CustomBean customBeanA(PasswordEncoder passwordEncoderA) {  // …  }  @Bean  public PasswordEncoder passwordEncoderA() {  return new BcryptPasswordEncoder();  }  }  @Configuration  public class ConfigB {  @Bean  public CustomBean customBeanB(PasswordEncoder passwordEncoderB) {  // …  }  @Bean  public PasswordEncoder passwordEncoderB() {  return new BcryptPasswordEncoder();  }  } |

Good Example

|  |
| --- |
| @Configuration  public class ConfigA {  @Bean  public CustomBean customBeanA(PasswordEncoder passwordEncoder) {  // …  }  }  @Configuration  public class ConfigB {  @Bean  public CustomBean customBeanB(PasswordEncoder passwordEncoder) {  // …  }  }  @Configuration  public class PasswordEncoderConfig {  @Bean  public PasswordEncoder passwordEncoder() {  return new BcryptPasswordEncoder();  }  } |

**Rule 4.3: Infrastructure Bean Grouping**

If you have multiple logically identical infrastructure beans, they should be grouped together in a single configuration.

|  |
| --- |
| @Configuration  public class ConfigA {  @Bean  public CustomBean customBeanA(@Qualifier(BCRYPT\_PASSWORD\_ENCODER) PasswordEncoder bcryptPasswordEncoder) {  // …  }  }  @Configuration  public class ConfigB {  @Bean  public CustomBean customBeanB(@Qualifier(NO\_OP\_PASSWORD\_ENCODER) PasswordEncoder noopPasswordEncoder) {  // …  }  }  @Configuration  public class PasswordEncoderConfig {  @Bean(BCRYPT\_PASSWORD\_ENCODER)  public PasswordEncoder bcryptPasswordEncoder() {  return new BcryptPasswordEncoder();  }  @Bean(NO\_OP\_PASSWORD\_ENCODER)  public PasswordEncoder noopPasswordEncoder() {  return new NoopPasswordEncoder();  }  } |

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.

|  |
| --- |
| **Rules of Thumb:**   * One **top-level dependent** per file. * Supporting beans stay in the same config if they serve only that top-level. * Shared beans move to a **common** or **infrastructure** config. * Infrastructure beans with variants belong in **one grouped config**. * If logic becomes **business-aware**, it belongs to an **application config or component**, not infra. |

### **5: [Mandatory] 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.

### **6: [Mandatory] Dependency Injection Rules**

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.

### 

### **8: [Recommended] Naming Convention**

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

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

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

## 

### 