Java Spring Framework - Java Based Config

Moving from XML-Based Configuration to Java-Based Configuration in Spring

- Until this point, the Spring project was configured using **XML-based configuration**.
- All beans managed by Spring were defined inside an XML file (commonly spring.xml or config.xml).
- The XML file name is flexible; if it changes, the reference in ClassPathXmlApplicationContext must be updated accordingly.

Why Java-Based Configuration?

- Many developers prefer **Java-based configuration** over XML.
- Reasons include:
    - XML being verbose and less readable for some developers
    - Preference for type safety and refactoring support in Java
- Spring supports:
    - **XML-based configuration**
    - **Java-based configuration**
    - **Annotation-based configuration** (to be covered later)
- In real projects, the configuration style depends on what the project already uses.

Replacing XML with Java Configuration

- Java-based configuration uses a **Java class** instead of an XML file.
- This Java class acts as a replacement for the XML configuration file.

Creating a Configuration Class

- A new package (e.g., config) is created to hold configuration-related classes.
- A new Java class is created (e.g., AppConfig).
- The class name is flexible and can be anything.

Changing the ApplicationContext Implementation

XML-Based Approach (Earlier)

```
ApplicationContext context =
    new ClassPathXmlApplicationContext("config.xml");
```

Java-Based Approach (Now)

```
ApplicationContext context =
    new AnnotationConfigApplicationContext(AppConfig.class);
```

- Both approaches use the **same Spring container**.
- The difference lies in **how the container is configured**:
  - XML-based → XML file
  - Java-based → Java class

Initial Issue with Java Configuration
- Attempting to retrieve a bean:

```
Desktop desk = context.getBean(Desktop.class);
```

- Results in an error:
  - **No qualifying bean of type Desktop available**
- Reason:
  - No configuration has been defined yet in AppConfig.

Enabling Java-Based Configuration with @Configuration
- To tell Spring that a class contains configuration:

```
@Configuration
```

- This annotation marks the class as a **Spring configuration class**.
- It is equivalent to the XML configuration file.

Defining Beans Using @Bean Annotation
- In XML, beans were defined using the <bean> tag.

- In Java configuration, beans are defined using the **@Bean** annotation.

Example: Defining a Desktop Bean

```
@Bean
public Desktop desktop() {
    return new Desktop();
}
```

- Key points:
  - The method returns the object to be managed by Spring.
  - The new keyword is used, but:
    - Spring calls this method
    - Spring creates, injects, and manages the object
  - The developer does not manually manage the lifecycle.

Complete Java Configuration Class

```
package org.springframeworkxmlbasedconfig.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframeworkxmlbasedconfig.Desktop;

@Configuration
public class AppConfig {

    @Bean
    public Desktop desktop() {
        return new Desktop();
    }
}
```

Retrieving the Bean from the Container

```
Desktop desk = context.getBean(Desktop.class);
desk.compile();
```

- The bean is retrieved **by type**, not by name.

- Spring searches for a bean of type Desktop and returns it.
- The method compile() executes successfully, confirming correct configuration.

Key Observations

- Java-based configuration:
  - Eliminates XML
  - Uses annotations like **@Configuration** and **@Bean**
- Spring still:
  - Creates the objects
  - Injects dependencies
  - Manages the lifecycle
- The developer only defines *how* the beans should be created.

Current Limitation

- No explicit bean name has been defined yet.
- Beans are currently accessed **by type only**.
- Bean naming and customization will be discussed in the next section.

Summary

- XML configuration can be fully replaced with Java configuration.
- AnnotationConfigApplicationContext is used instead of ClassPathXmlApplicationContext.
- @Configuration replaces the XML file.
- @Bean replaces the <bean> tag.
- Even though new is used, Spring remains responsible for object creation and management.

Understanding Bean Names in Java-Based Spring Configuration

Why Bean Names Matter in Java Configuration

- When retrieving a bean using:

```
context.getBean(Desktop.class);
```

the lookup is done **by type**, not by name.

- The question arises:
  - What if we want to retrieve a bean **by name**, similar to how it worked in XML configuration?

Attempting to Use an Explicit Bean Name

- Example attempt:

Desktop dt = context.getBean("com2", Desktop.class);

- Result:
  - Runtime error: **No bean named 'com2' available**
- Reason:
  - Unlike XML configuration, the bean name has **not been explicitly defined** yet in Java configuration.

Default Bean Naming in Java-Based Configuration

- In Java configuration, when using @Bean:
  - **The default bean name is the method name**
- Example:

```
@Bean
public Desktop desktop() {
    return new Desktop();
}
```

- Default bean name:
  - "desktop"

Retrieving the Bean Using the Default Name

Desktop dt = context.getBean("desktop", Desktop.class);

- This works because "desktop" matches the method name.

Customizing the Bean Name Using @Bean

- You can explicitly set a bean name using the name attribute of @Bean.

Example: Single Custom Name

```
@Bean(name = "com2")
public Desktop desktop() {
    return new Desktop();
}
```

- Now the bean can be retrieved as:

Desktop dt = context.getBean("com2", Desktop.class);

Assigning Multiple Names (Aliases) to a Bean

- Spring allows **multiple names (aliases)** for the same bean.

Example: Multiple Names

```
@Bean(name = {"com2", "mac", "dell"})
public Desktop desktop() {
    return new Desktop();
}
```

- The same bean instance can be retrieved using any of these names:
    - "com2"
    - "mac"
    - "dell"
- If a name not listed here is used, Spring will throw a **No bean named … available** error.

Using the Default Bean Name Again

- If no name attribute is provided:

```
@Bean
public Desktop desktop() {
    return new Desktop();
}
```

- The default bean name reverts to:
    - "desktop"

Valid Retrieval

Desktop dt = context.getBean("desktop", Desktop.class);

Key Differences from XML Configuration

- **XML Configuration**
    - Bean name is explicitly defined using the id attribute.
- **Java Configuration**
    - Bean name defaults to the **method name**.
    - Can be overridden using @Bean(name = "...").

Important Observations

- Bean lookup can be done:
    - By **type**
    - By **name + type**
- Naming decisions depend on:
    - Project conventions
    - Clarity and avoidance of ambiguity
- Multiple names are optional but supported.

What's Next

- By default, all beans are **singleton**.
- Upcoming topics:
    - Using **prototype scope** in Java-based configuration
    - Making a bean **primary** when multiple beans of the same type exist

---

Summary

- Default bean name in Java-based configuration = **method name**

- Custom bean names can be defined using @Bean(name = "...")
- Multiple aliases can be assigned to a single bean
- Bean retrieval by name behaves similarly to XML once names are defined
- Scope and primary configuration will be covered next

Bean Scope in Java-Based Spring Configuration (Singleton vs Prototype)

Default Bean Scope: Singleton

- By default, **every Spring bean is a singleton**.
- Meaning:
  - When the application starts, the **IOC container is created**.
  - A **single instance** of each singleton bean is created and stored in the container.
- In the current setup:
  - Only the **Desktop** bean exists.
  - **Alien** and **Laptop** beans are not configured or used.
  - Therefore, **only one object** is created.

Key Observation

- Output confirms:
  - Only **one Desktop object** is created.
  - No logs for Alien or Laptop object creation.

Retrieving the Same Bean Multiple Times (Singleton Behavior)

Code Example

```
Desktop dt1 = context.getBean(Desktop.class);
dt1.compile();

Desktop dt2 = context.getBean(Desktop.class);
dt2.compile();
```

Result

- Output:

```
Desktop Object Created...
Compiling in Desktop....
Compiling in Desktop....
```

- Explanation:
  - getBean() is called twice.
  - **Same Desktop instance** is returned both times.
  - Constructor runs only once.
  - Method compile() runs twice on the same object.

Requirement: Creating Multiple Objects (Prototype Scope)

- Sometimes, a single shared instance is **not desired**.
- Requirement:
  - Each call to getBean() should return a **new object**.
- This behavior is achieved using **prototype scope**.

Using Prototype Scope in Java-Based Configuration

Annotation Used

- @Scope

Default Value

- singleton

Changing Scope to Prototype

```
@Bean
@Scope(value = "prototype")
public Desktop desktop() {
    return new Desktop();
}
```

- This is equivalent to:

```
scope="prototype"
```

in XML configuration.

Prototype Behavior in Action

Code

```
Desktop dt1 = context.getBean(Desktop.class);
dt1.compile();

Desktop dt2 = context.getBean(Desktop.class);
dt2.compile();
```

Output

```
Desktop Object Created...
Compiling in Desktop....
Desktop Object Created...
Compiling in Desktop....
```

Explanation

- Each getBean() call:
  - Creates a **new Desktop object**
- Constructor runs **every time**
- Objects are **not shared**

Comparison: Singleton vs Prototype

Aspect	**Singleton**	**Prototype**
Default scope	Yes	No
Objects created	One	New object per request
Constructor called	Once	Every time
Managed by Spring	Yes	Yes (creation only)

Important Notes

- Singleton:
  - Best for shared, stateless components.
- Prototype:
  - Useful when object state should not be shared.

- Scope is defined:
    - In **XML** using scope="prototype"
    - In **Java config** using @Scope("prototype")

Key Takeaway

- By default, Spring beans are **singleton**.
- To get **multiple instances**, explicitly set:

```
@Scope("prototype")
```

- Java-based configuration provides the same flexibility as XML, using annotations instead of tags.

Title: Creating and Configuring Spring Beans Using Java-Based Configuration

- Demonstrates creating Spring beans using **Java-based configuration** instead of XML.
- Focuses on defining beans for Alien and Desktop classes.
- Shows how Spring manages object creation and dependency injection through @Bean.

Title: Defining an Alien Bean Without Explicit Bean Name

- The Alien bean is created without specifying a bean name, relying only on the **class type**.
- Initial configuration skips text-related properties and focuses on:
    - age
    - Computer dependency (comp)
- Bean scope is changed from **prototype** to **simple (singleton)**.

Title: Understanding “No Qualifying Bean” Error

- Runtime error: **No qualifying bean of type Alien**.
- Cause:
    - Alien was not defined as a bean in AppConfig.
- Resolution:

    - Create a method in AppConfig that:
        - Returns an Alien object
        - Is annotated with @Bean

Title: Assigning Properties to Bean Using AppConfig

- Instead of setting values directly in the main method:
    - Alien properties are assigned inside the @Bean method.
- Example:
    - age is set using obj.setAge(25) inside AppConfig.
- Result:
    - getAge() correctly returns the value from configuration.

Title: NullPointerException Due to Missing Dependency Injection

- Error encountered when calling obj.code():
    - comp (Computer dependency) is null.
- Reason:
    - Unlike XML configuration, no reference was established between Alien and Desktop.
- Insight:
    - code() depends on Computer.compile(), which fails if comp is not injected.

Title: Manually Injecting Desktop into Alien Bean

- Desktop is already defined as a bean and implements Computer.
- Dependency injection done explicitly:
    - obj.setComp(desktop())
- Outcome:
    - Application runs successfully.
    - compile() method executes from Desktop.

Title: Problem of Tight Coupling

- Directly injecting desktop() into Alien causes **tight coupling**.
- Issue:
    - If another implementation (e.g., Laptop) is introduced, code must change.
- This violates flexibility and scalability principles.

Title: Constructor-Based Dependency Injection Using Interface

- Solution:
  - Inject dependency using the Computer interface instead of Desktop.
- Spring behavior:
  - Detects Alien depends on Computer.
  - Searches the container for a bean implementing Computer.
- Result:
  - Spring automatically injects the appropriate implementation.

Title: Autowiring Behavior in Java Configuration

- Method parameter injection is used:

```
@Bean
public Alien alien(@Autowired Computer com)
```

- Key points:
  - @Autowired is optional in newer Spring versions.
  - Spring resolves the dependency by type.
- Creates a loose coupling between Alien and Computer.

Title: Multiple Beans of Same Type and Ambiguity

- Scenario introduced:
  - Multiple beans implementing Computer (e.g., Desktop, Laptop).
- Question raised:
  - Which bean will Spring choose?
- Clarification:
  - This ambiguity was handled earlier in XML using primary.
  - Behavior in Java-based configuration will be explained next.

Title: Comparison with XML-Based Autowiring

- In XML configuration:
  - autowire="byType" was used.
  - Explicit references override primary.

- Key takeaway:
    - primary is only used when Spring faces ambiguity.
    - Explicit references always take precedence.

Title: Final Execution Flow Summary

- Beans created:
    - Desktop (implements Computer)
    - Alien (depends on Computer)
- Spring actions:
    - Creates Desktop bean.
    - Injects it into Alien.
- Output confirms:
    - Objects created successfully.
    - age is printed.
    - compile() executes without error.

Title: Introducing Multiple Beans of the Same Interface (Desktop and Laptop)

- Until now, only one Computer implementation (Desktop) existed in the Spring container.
- A new bean for Laptop is added to the Java-based configuration.
- Both Desktop and Laptop implement the same interface: **Computer**.
- A new @Bean method is created:
    - Method returns a Laptop object.
    - Method name can be anything, commonly same as class name.
    - Package for Laptop must be imported.

Title: Error Caused by Multiple Beans of the Same Type

- Application fails at runtime with error:
    - **Expected single matching bean but found two: desktop, laptop**
- Reason:
    - Spring tries to inject a Computer dependency into Alien.
    - Two beans of type Computer exist (Desktop and Laptop).
- Spring cannot decide which one to inject automatically.

Title: Understanding Bean Ambiguity in Dependency Injection

- When a dependency is injected by **type**, Spring expects:
  - Exactly one matching bean.
- If more than one bean matches:
  - Spring throws a **NoUniqueBeanDefinitionException**.
- This behavior is the same as what happens in **XML configuration**.

Title: Resolving Ambiguity Using @Qualifier

- **@Qualifier** allows explicit selection of the required bean.
- Works similar to the ref attribute in XML configuration.
- Usage:
  - Specify the **bean name** inside @Qualifier.
  - Example: @Qualifier("desktop")
- Important notes:
  - Bean name must match exactly.
  - Incorrect names (e.g., desktop1) will cause errors.
- Result:
  - Spring injects the specified bean without confusion.

Title: Resolving Ambiguity Using @Primary

- Alternative solution: use **@Primary**.
- Applied directly on one bean definition.
- Example:

```
@Bean
@Primary
public Laptop laptop() {
   return new Laptop();
}
```

- Behavior:
  - When multiple beans of the same type exist:
    - Spring chooses the bean marked with @Primary.
- No need to modify the injection point.

Title: Qualifier vs Primary – Key Differences

- **@Qualifier**
  - Explicitly specifies which bean to inject.
  - Used at the injection point.
  - Similar to ref in XML.
- **@Primary**
  - Defines a default bean when multiple options exist.
  - Used at the bean definition level.
- Both approaches resolve dependency conflicts effectively.

Title: Alien Bean with Interface-Based Injection

- Alien bean depends on the Computer interface:

```
@Bean
public Alien alien(Computer com) {
   Alien obj = new Alien();
   obj.setAge(28);
   obj.setComp(com);
   return obj;
}
```

- Dependency resolution:
  - Controlled by either @Qualifier or @Primary.
- Maintains **loose coupling** between Alien and concrete implementations.

Title: Key Takeaways from Multiple Bean Configuration

- Multiple beans implementing the same interface cause ambiguity.
- Spring provides two main solutions:
  - **@Qualifier** for explicit selection.
  - **@Primary** for default preference.
- Java-based configuration mirrors XML behavior closely.
- This concept becomes more common and important in **Spring Boot** applications.

Title: Default Values When Using @Component Without Injection

- When using **@Component**, Spring creates the object using the **default constructor**.
- As a result:
  - age gets the default value **0**
  - comp (Computer reference) is **null**
- Output confirms this:
  - Age printed as 0
  - NullPointerException when calling compile() on comp
- Reason:
  - Spring created the Alien object but **did not inject dependencies**.

Title: Solving Dependency Injection Using @Autowired

- **@Autowired** tells Spring:
  - Look inside the container
  - Find a matching bean
  - Inject it automatically
- When applied to Computer comp:
  - Spring searches for a bean of type **Computer**
  - Injects it into Alien

```
@Autowired
private Computer comp;
```

- This fixes the null issue **only if exactly one Computer bean exists**.

Title: Problem with Multiple Implementations of the Same Interface

- Two classes implement Computer:
  - Desktop
  - Laptop
- When both are annotated with **@Component**:
  - Spring finds **two beans**
- Error:
  - *Expected single matching bean but found 2: desktop, laptop*
- Spring does not know which one to inject.

Title: Resolving Multiple Bean Conflicts Using @Qualifier

- **@Qualifier** specifies **which bean to inject by name**.
- Bean name rules when using @Component:
  - Default bean name = class name with **first letter lowercase**
  - Desktop → desktop
  - Laptop → laptop

```
@Autowired
@Qualifier("laptop")
private Computer comp;
```

- This explicitly tells Spring which implementation to inject.

Title: Custom Bean Names with @Component

- You can override the default bean name:

```
@Component("computer2")
public class Desktop implements Computer { }
```

- Then reference it using @Qualifier:

```
@Autowired
@Qualifier("computer2")
private Computer comp;
```

- If names do not match:
  - Spring throws a **No qualifying bean** error.

Title: Alternative to @Qualifier – @Primary

- **@Primary** marks one bean as the default choice.
- Used when multiple beans of the same type exist.
- Example:

```
@Component
@Primary
public class Laptop implements Computer { }
```

- Spring will inject Laptop automatically **without @Qualifier**.
- Works only when there is **confusion** between multiple beans.

Title: Types of Dependency Injection in Spring

Spring supports **three types of injection** using @Autowired:

1. Field Injection

- Injection directly on the field.
- Used in this example.

```
@Autowired
private Computer comp;
```

---

2. Constructor Injection

- Injection through constructor parameters.
- @Autowired placed on constructor.

```
@Autowired
public Alien(Computer comp) {
    this.comp = comp;
}
```

---

3. Setter Injection (Preferred Alternative)

- Injection through setter method.
- @Autowired placed on setter.

```
@Autowired
public void setComp(Computer comp) {
    this.comp = comp;
}
```

- Recommended when not using constructor injection.

Title: Summary of the Fix

- **Problem**: Objects created, but dependencies not injected.

- **Cause**: Missing autowiring when using @Component.
- **Solutions**:
  - Use **@Autowired** to inject dependencies
  - Resolve conflicts using:
    - **@Qualifier** (explicit choice)
    - **@Primary** (default preference)
- Injection methods:
  - Field
  - Constructor
  - Setter

Title: Resolving Multiple Bean Conflicts Using @Primary

- When multiple beans implement the same interface (e.g., Desktop and Laptop implementing Computer), Spring throws an error:
  - *Expected single matching bean but found 2*
- One way to resolve this is by using **@Primary**.
- **@Primary** tells Spring:
  - "If there is a confusion, prefer this bean by default."

```
@Component
@Primary
public class Desktop implements Computer {
}
```

- With this configuration:
  - Whenever Spring needs a Computer bean
  - And no further instruction is given
  - **Desktop** will be injected automatically.

Title: Using @Qualifier Along with @Primary

- **@Qualifier** explicitly specifies which bean should be injected.
- Example using setter injection:

```
@Autowired
@Qualifier("laptop")
public void setComp(Computer comp) {
   this.comp = comp;
```

}

- Default bean names when using @Component:
  - Desktop → desktop
  - Laptop → laptop

Title: Priority Between @Qualifier and @Primary

- When **both @Primary and @Qualifier are used together**:
  - **@Qualifier takes precedence over @Primary**
- Scenario:
  - Desktop is marked as @Primary
  - @Qualifier("laptop") is used during injection
- Result:
  - **Laptop** is injected, not Desktop
- Conclusion:
  - Explicit instruction (@Qualifier) always overrides default preference (@Primary)

Title: When to Use @Primary vs @Qualifier

- **Use @Primary**:
  - When one implementation should be the default choice
  - When you want to avoid specifying bean names everywhere
- **Use @Qualifier**:
  - When you need precise control
  - When different consumers need different implementations
- Both are valid and commonly used together depending on design needs.

Using @Scope and @Value Annotations in Spring

---

1. Managing Bean Scope with @Scope

- Spring beans can have different **scopes**, commonly:
  - **Singleton**: Only one instance is created per Spring container

(default).
    - **Prototype**: A new instance is created each time the bean is requested.
- To set a bean's scope in Java-based configuration or with annotations:

```
@Component
@Primary
@Scope("prototype") // This bean will be prototype scoped
public class Desktop implements Computer {
}
```

- **Usage**:
    - When Spring creates a bean marked with @Scope("prototype"), a new instance is created every time it is requested from the container.
    - @Scope can be applied to any component-managed class.

---

2. Injecting External Values with @Value

- **@Value** is used to assign values to fields from external sources such as property files, or inline literals.
- Difference between assigning a value directly vs using @Value:
    - Direct assignment (e.g., private int age = 28;) **hardcodes the value**.
    - Using @Value allows **externalization** and **flexibility**, enabling values to come from:
        - Property files
        - Environment variables
        - Expression evaluation

```
@Component
@Primary
@Scope("prototype")
public class Desktop implements Computer {

   @Value("28") // Injects value 28 into age
   private int age;

   // other methods...
}
```

- **Advantages**:
    - Centralized configuration
    - Easy to change values without modifying source code

- Supports dynamic injection from configuration files

---

3. Key Points

- **@Scope**: Controls lifecycle of Spring beans (singleton vs prototype).
- **@Value**: Injects literal values or property values into fields.
- Can be combined with other stereotype annotations like @Component and @Primary.
- Helps decouple hardcoded data and makes beans flexible and configurable.