Java configuration annotations

You're developing a Spring Boot app using **Java configuration annotations** — like @PostConstruct, @Inject, etc.

But… these annotations are actually part of **Java’s standard libraries** (called JSE or JEE modules), not created by Spring.

**🚀 So how do they work?**

These annotations are just **tags** or **instructions** on your code.

But those tags won’t do anything by themselves — your **framework (like Spring, Hibernate, or Servlet)** is the one that looks at them and decides what to do.

**🔧 Different frameworks act on them differently:**

* In **Spring**, the Spring Framework reads the annotations and does things like injecting objects or calling methods.
* In **Hibernate**, the Hibernate engine might use them for DB entity lifecycle.
* In **Servlet**, the server might use them to manage servlet lifecycle.

**📚 Let’s break down the examples:**

| **Annotation** | **What it does** | **Like in Spring** |
| --- | --- | --- |
| @PostConstruct | Run this method **after the bean is created and dependencies are injected**. | Like init() method |
| @PreDestroy | Run this method **just before the bean is destroyed or app shuts down**. | Like destroy() method |
| @Inject | **Inject a dependency by type**. | Like @Autowired |
| @Named("myBean") | **Give a name to a bean** or **inject a bean using its name**. | Like @Component("myBean") or @Qualifier("myBean") |
| @Resource(name="myBean") | **Inject a bean by name first**, then fallback to type if name not found. | Like @Autowired + @Qualifier |

**🧠 Simple analogy**

Think of annotations as **stickers** you place on methods or fields:

* 📌 @PostConstruct = “Run this once when ready!”
* 📌 @PreDestroy = “Do this before shutdown!”
* 📌 @Inject = “Give me a helper object automatically.”
* 📌 @Named = “This is my official name.”
* 📌 @Resource = “Find this exact helper by name.”

But who follows the stickers? → **Spring does**, if you’re in a Spring Boot app.

**🎯 Final Words**

You don’t have to use these — Spring has its own versions like @Autowired, @Component, etc.  
But these **standard annotations** work across many Java platforms, so Spring supports them too.

## ✅ **Spring Annotations are used more in practice**

| **Reason** | **Explanation** |
| --- | --- |
| 🧩 **Simpler and more powerful** | Spring annotations like @Component, @Autowired, @Service, etc., are tightly integrated with Spring’s full feature set. |
| 📚 **More documentation and tutorials** | Most examples, courses, and real-world projects use Spring’s annotations. |
| ⚙️ **Better support for advanced features** | Spring annotations support things like profiles, conditional beans, lifecycle hooks, scopes, etc. |
| 🛠️ **Easier debugging and customization** | Spring provides more control using annotations like @Qualifier, @Value, @Configuration, etc. |

## 🎯 Conclusion

**In Spring-based applications, 95%+ of the time, developers prefer and use Spring annotations.**

Java standard annotations are still supported by Spring and used in some cases (especially @PostConstruct), but **Spring's own annotations dominate in the real world**.

### ✅ ****Dependency Injection through**** application.yml ****– Notes****

1. **Industry Use:**
   * Widely used in Spring Boot microservices to inject configuration like database URLs, service endpoints, credentials, and feature flags.
   * Helps manage multiple environments (dev, test, prod) using Spring Profiles (application-dev.yml, etc.).
2. **Advantages:**
   * **Externalized & Environment-Specific Config:** You can keep your configuration outside the codebase and easily switch between environments.
   * **Readable & Structured Format:** YAML is cleaner and more readable than .properties for nested configuration.
   * **Loose Coupling:** Injects configuration into beans via @ConfigurationProperties, avoiding hardcoding and improving flexibility.

Eg: SpringBootDependencyInjectionUsingYml

### ✅ Profiles in Spring Boot – Summary Notes

#### 🔹 Support Across All Spring Variants:

Spring and Spring Boot **support environment-specific profiles**. Profiles can be used in:

* a. XML-driven configuration
* b. Annotation-driven configuration
* c. 100% Java code-based configuration
* d. Spring Boot (via .properties or .yml files)

#### 🔹 What is an Environment?

An environment is a complete setup required to run an application:

**Code + Server + Database + JARs + Config (.properties/.yml)**

### 🔹 Why Use Profiles?

When we run our app in different places like **development**, **testing**, or **production**, we need different settings.

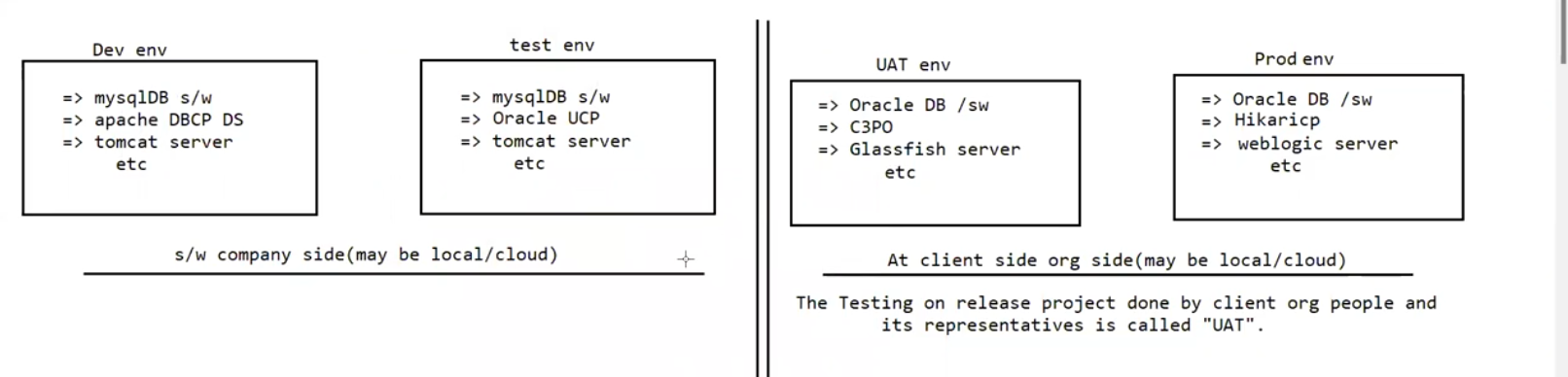
For example:

* The **database** might be different in each place
* We may want **more or less logging**
* **API links** can change
* **Passwords** are also different

Instead of changing these settings again and again, we create separate **profiles**.  
Each profile has its own settings, and Spring Boot picks the right one automatically.

#### 🔹 Common Profiles in Real-Time Projects:

* 🛠 **Dev profile** – For development setup
* 🧪 **Test profile** – For unit/integration testing
* 🧾 **UAT profile** – For User Acceptance Testing
* 🚀 **Prod profile** – For production deployment



### 🔹 Different Environments in a Project (Simple Explanation)

When building software, we use **different setups** (called environments) for each stage of development:

#### ✅ **1. Dev Environment (Developer's side)**

* Used by developers to build and test code
* Uses:
  + MySQL Database
  + Apache DBCP (DataSource)
  + Tomcat Server

#### ✅ **2. Test Environment (QA team side)**

* Used by testers to check if the app works correctly
* Uses:
  + MySQL Database
  + Oracle UCP (Connection Pooling)
  + Tomcat Server

#### ✅ **3. UAT Environment (Client's side – User Acceptance Testing)**

* Client tests if the software works as expected
* Uses:
  + Oracle Database
  + C3P0 (Connection Pool)
  + GlassFish Server

#### ✅ **4. Prod Environment (Live for real users)**

* Real users use the app here
* Uses:
  + Oracle Database
  + HikariCP (High-performance DB connection)
  + WebLogic Server

### 🔹 Key Note:

* **Dev/Test**: Used inside the software company (local/cloud)
* **UAT/Prod**: Used by the **client organization**

💡 UAT = Client checks and approves the final software before it goes liv

### @Profile in Spring Boot

#### 🔹 What is @Profile?

* @Profile is used to **control which beans are loaded** in a specific environment (like dev, test, prod).
* It helps in **environment-specific bean creation**.

#### 🔹 Key Points:

1. @Profile("dev") → Bean loads only when dev profile is active.
2. @Profile({"dev", "test"}) → Bean loads in either dev or test profile.
3. If **no @Profile is used**, the bean is available in **all profiles**.
4. Can be used on classes annotated with:
   * @Component
   * @Service
   * @Repository
   * @Configuration

### 🔍 Where @Profile is commonly used:

| **Annotation** | **Use Case** |
| --- | --- |
| @Component | Generic Spring-managed bean |
| @Service | Business logic/service layer beans |
| @Repository | DAO layer beans |
| @Configuration | Java-based configuration classes with @Bean methods |

1. Helps avoid writing if-else logic in code for different environments.

#### ✅ Example:

@Profile("dev")

@Component

public class DevDataSourceConfig {

// This bean will be active only in the 'dev' profile

}

Or for multiple profiles:

@Profile({"dev", "test"})

@Service

public class DevOrTestService {

// This bean works in both 'dev' and 'test' profiles

}

#### 🔸 Note:

If you **don't use @Profile**, then the bean will be available in **all environments** by default.

### 📘 Notes: Using @Profile on @Bean Methods in Spring Boot

#### 🔹 What it means:

If you're using a @Configuration class and want to create beans with the @Bean method that should work **only in certain environments**, you can use @Profile **on top of the @Bean method**.

#### ✅ Key Points:

1. @Profile can be applied to individual @Bean methods.
2. This helps load only the necessary bean depending on the active profile.
3. Keeps configuration clean and profile-specific without manual checks.
4. Useful when the same class returns different beans for different profiles.

#### ✅ Example:

@Configuration

public class PersistenceConfig {

@Bean("dsDBCP2")

@Profile("dev")

public DataSource createUsingDBCP2() {

// Return a DBCP2 DataSource for dev

}

@Bean("dsOraUCP")

@Profile("dev")

public DataSource createUsingOracleUCP() {

// Return Oracle UCP DataSource for dev

}

}

🔸 These beans (dsDBCP2, dsOraUCP) will only be created when **dev** profile is active.

### 📘 Notes: Profile-Specific Properties/YML Files in Spring Boot

#### 🔹 Why Separate Files?

To keep environment settings clean and organized, Spring Boot allows you to create **profile-specific configuration files**.

### ✅ File Naming Format:

application.properties → Default (used if no profile is active)

application-dev.properties → For 'dev' profile

application-uat.properties → For 'uat' profile

application-pro.properties → For 'prod' profile

application-<profile>.yml → Same works for YAML format

You can define different DB URLs, logging levels, ports, etc. in each file.

### 🔹 How to Activate a Specific Profile?

#### a. In application.properties or application.yml

properties

spring.profiles.active = dev

#### b. Using JVM/System Property (Command Line)

bash

java -jar app.jar -Dspring.profiles.active=dev

### ✅ Summary Points:

* Spring Boot loads profile-specific .properties or .yml file based on the **active profile**.
* Useful for changing behavior/settings across **dev**, **test**, **uat**, and **prod** environments.
* Helps avoid manual changes before deployment.

In a YAML file (like application.yaml in Spring Boot), **profiles** are used to manage different configurations for different environments—such as **development**, **testing**, **staging**, and **production**. This is a core feature in **Spring Boot** that allows you to isolate environment-specific settings cleanly.

**🔧 Why use profiles?**

Each environment (dev, test, prod, etc.) might have:

* Different **database URLs**
* Different **logging levels**
* Different **API endpoints**
* Different **feature toggles**

Instead of maintaining separate files, you can include all configuration yml files in one yml file

**✅ Basic Syntax**

yaml

spring:

profiles:

active: dev # Sets the active profile to "dev"

---

spring:

profiles: dev # Block for dev profile

server:

port: 8081

servlet:

context-path: /dev-app

---

spring:

profiles: prod # Block for prod profile

server:

port: 80

servlet:

context-path: /prod-app

* The --- separates profile-specific blocks.
* spring.profiles sets the name of the profile that block belongs to.
* spring.profiles.active tells Spring Boot which profile to use.

**🧪 How to Activate a Profile**

1. **In application.yaml itself:**

spring:

profiles:

active: dev

1. **As a Command Line Argument:**

bash

java -jar myapp.jar --spring.profiles.active=prod

1. **As an Environment Variable:**

bash

export SPRING\_PROFILES\_ACTIVE=prod

1. **In application.properties (if using instead of YAML):**

properties

spring.profiles.active=dev

**🧩 Example with Database Configuration**

yaml

---

spring:

profiles: dev

datasource:

url: jdbc:mysql://localhost:3306/dev\_db

username: dev\_user

password: dev\_pass

---

spring:

profiles: prod

datasource:

url: jdbc:mysql://prod-host:3306/prod\_db

username: prod\_user

password: prod\_pass

**✅ Best Practices**

* Use application.yaml for default settings and override them in profile-specific blocks.
* Avoid hardcoding spring.profiles.active inside YAML for production. Prefer command-line or environment variable overrides.
* Sensitive values (like passwords) should ideally come from external sources or secrets vaults, not YAML.

**✅ How Spring Boot maps spring.profiles.active: dev to application-dev.yml**

In Spring Boot, there's a **convention-based mechanism** for loading profile-specific properties.

**🔍 How It Works**

1. **application.yml (or .properties) is the base configuration.**
2. If it contains:

spring:

profiles:

active: dev

...this tells Spring Boot to activate the dev profile.

1. **Spring Boot will then automatically look for**:
   * application-dev.yml or application-dev.properties

**🧠 Spring Boot Logic (Internally)**

Spring Boot uses the spring.profiles.active property to determine the **active profile(s)**. When it sees dev is active, it will look for a matching config file using this naming pattern:

application-{profile}.yml

So in your case:

spring.profiles.active: dev

➡ Spring Boot loads:

* application.yml (base config)
* application-dev.yml (overrides base for "dev" profile)

**💡 Summary**

* You **don’t** need to mention application-dev.yml directly anywhere.
* The framework uses the naming convention automatically.
* In code, @Profile("dev") works because Spring knows the active profile from the config.

**🛠️ Bonus: Multiple profiles**

You can also activate multiple profiles:

spring:

profiles:

active: dev,test

Then both application-dev.yml and application-test.yml will be loaded, with test taking precedence over dev if there are overlaps.