**1. Spring (the “original” framework)**

* **What it is:** A general Java framework (your toolbox) for building apps.
* **Dependency Injection:** You explicitly tell Spring which objects (beans) to create and how to wire them together.
* **Configuration styles:** You can choose:
  1. XML files
  2. XML + Java annotations
  3. Pure Java code (no XML at all)
* **Bootstrapping:** You manually create the application context (IOC container) in your code.
* **Web & data:**
  1. **No embedded web server** (you must deploy your WAR to Tomcat, Jetty, etc.)
  2. **No built-in in-memory database** (you configure your own)
* **Weight & scope:**
  1. **Lightweight** (nothing happens until you wire it up)
  2. **Not opinionated for microservices** (you’d assemble most pieces yourself)

**2. Spring Boot (the “opinionated” starter kit)**

* **What it is:** A layer on top of Spring that “just works”—it gives you a ready-made app.
* **Auto-configuration:** Spring Boot “guesses” the common beans you need (datasources, MVC setup, etc.) so you rarely write boilerplate.
* **Configuration style:**
  + No XML—everything comes from sensible defaults + your application.properties / application.yml.
* **Bootstrapping:**
  + Just call

SpringApplication.run(MyApp.class, args);

* + The IOC container is created for you automatically.
* **Web & data:**
  + **Embedded web server** (Tomcat/Jetty built-in—just run the JAR)
  + **Embedded in-memory DB** (H2 by default for quick testing)
* **Weight & scope:**
  + **Heavier** because it brings in a lot of auto-configured components
  + **Designed for microservices** (built-in support for standalone services, health checks, metrics, easy Dockerization, etc.)

## 1. Use @Import when…

**“I have a whole bundle of beans living elsewhere—bring them into my app.”**

### Simple terms

* **Beans** = little pieces of code (services, helpers) that Spring can run.
* **Elsewhere** = another package, another module, or a library jar—not in your usual folder.

### Why you need it

1. **Group loading**  
   You’ve collected related beans together in one @Configuration class. You don’t want to register each one by hand.
2. **Keep code tidy**  
   The external beans stay in their own place. You just import them, instead of copying files around.
3. **Turn features on/off**  
   Want the audit feature? Add @Import(AuditConfig.class). Don’t want it? Remove that line.

### How it looks in code

// AuditConfig.java (in a different package or jar)

@Configuration

public class AuditConfig {

@Bean

public AuditService auditService() {

return new AuditService();

}

}

// MyApp.java (your main app)

@SpringBootApplication

@Import(AuditConfig.class) // ← brings in the whole AuditConfig bundle

public class MyApp { … }

## 2. Use @Autowired when…

**“I need one of those beans—that service/tool/helper—in my class.”**

### Simple terms

* **Inject** = hand it over. You don’t build it yourself; Spring gives it to you.
* **Point of use** = inside your service, controller, or any other bean.

### Why you need it

1. **Easy wiring**  
   No more getBean(...) calls. Just annotate and Spring fills it in.
2. **Cleaner code**  
   Your class just lists what it needs; it doesn’t fetch or create it.
3. **Swap in tests**  
   In unit tests, you can replace that bean with a fake or mock—Spring injects whatever you configure.

### How it looks in code

@Service

public class ReportService {

private final AuditService auditService;

@Autowired // ← Spring injects the AuditService here

public ReportService(AuditService auditService) {

this.auditService = auditService;

}

public void runReport() {

auditService.record("Report run");

}

}

## Putting it together

1. **@Import**
   * **Role**: Register a whole set of beans you didn’t write in your main app.
   * **Think**: “Bring this box of tools into my workshop.”
2. **@Autowired**
   * **Role**: Inside your class, ask Spring: “Give me that one tool (bean) I need.”
   * **Think**: “Pass me the hammer from my toolbox.”

Eg: SpringBootConfigurationProperties

What is the difference b/w @Value and @CofigurationProperties?

@Value

=> It is given by Spring framework, so it can be used in Spring and SpringBoot applications.

=> Support single value injection to Spring bean property.

=> It performs field level injection(setters not required)

=> Common prefix of all keys are not required in application.properties/application.yml file

=> Keys in properties file and property names need not match.

=> If specified key is not present then it would result in "IllegalArgumentException".

@ConfigurationProperties

=> It is given by SpringBoot framework, so it can be used only SpringBoot applications.

=> Support bulk operation

=> It perform setter level injection internally, so setters are mandatory

=> Common prefix of all keys are required in application.properties/application.yml file.

=> keys in properties file and property names should match

=> If the matching key is not found then it would neglect the injection.

Note: While working with @ConfigurationProperties, it is always suggested to add configuarationProcessor inside pom.xml file

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-configuration-processor</artifactId>

<optional>true</optional>

</dependency>

Note:

If we try to inject different values to spring bean property using both Fieldlevel(@value) and @ConfigurationProperties annotations, which one will be injection?

Answer :: Since @ConfigurationProperties uses setter injection, so the values injected at field level(@Value) will be overriden with Setter level

Eg: SpringBootBeanInjectionWithConfigurationProperties

Eg: SpringBootBeanInjectionWithCollectionProperties

 **What “YAML” means**

* Originally it was called “Yet Another Markup Language,” but today we think of it as a human-friendly data format (it’s not really just “markup”).

 **File extensions**

* You’ll see YAML files named either something.yml or something.yaml.

 **Why YAML beats .properties for nested data**

* **Properties files** are just flat key=value pairs.

properties

server.port=8080

server.host=example.com

* If you have deeper structure (lists, objects inside objects), you end up repeating long prefixes over and over:

properties

user.address.street=123 Elm St

user.address.city=Springfield

user.address.zip=12345

* **YAML** uses indentation to show nesting, so it’s much shorter and clearer:

yaml

user:

address:

street: 123 Elm St

city: Springfield

zip: 12345

* This clean nesting is exactly what **@ConfigurationProperties** in Spring Boot loves—it can “bulk-inject” entire groups of settings into one Java object.

 **Spring Framework vs. Spring Boot support**

* Plain **Spring Framework** (the core libraries) only knows about .properties files by default.
* **Spring Boot** adds built-in support for reading YAML, so you can drop a application.yml in your project and Spring Boot will load it automatically.

 **Under the hood: SnakeYAML**

* When Spring Boot reads your application.yml, it actually calls the **SnakeYAML** library (a popular open-source parser) to turn your indented text into Java maps, lists, and primitives.

 **Don’t repeat the same key twice at one level**

yaml

# ❌ WRONG: “server:” is used twice under the same indent

server:

port: 8080

server:

host: example.com

# ✅ RIGHT: combine under one “server:” block

server:

port: 8080

host: example.com

 **Use colons (:) instead of dots (.) to start a nested block**

* If your old .properties looked like db.user.name=alice, in YAML you turn each “.” into its own indented level:

yaml

db:

user:

name: alice

* You must indent child lines by at least one space (two spaces is common).

 **Values are always key: value (not =)**

yaml

# ❌ WRONG

timeout=30

# ✅ RIGHT

timeout: 30

 **Lists (arrays, sets) start each item with a dash (-)**

yaml

fruits:

- apple

- banana

- cherry

 **Maps (dictionaries) and “has-a” sub-objects become nested blocks**

* When one setting naturally contains several smaller settings, turn it into its own map:

yaml

company:

name: In.Orcas

address:

street: 123 Elm St

city: Springville

 **Comments start with #**

yaml

# This line is ignored by the parser

debug: true # you can also end-of-line comment

Eg: SpringBootBeanInjectionWithCollectionYml