Preface — what this guide is (and isn't)

- **This is not** a Spring Boot tutorial; it's the **Spring Core foundation** you need to understand *why* Spring Boot does what it does.
- I assume you know Java (classes, interfaces, annotations, exceptions, basic I/O, Maven/Gradle).
- I'll define every new term before using it. If you see a concept you don't know, I explain it right away.
- Small runnable code snippets are included (use Java 17+ and Spring Boot 3+ if you want to run examples).

Basic terms (definitions first — read these once and refer back)

- **Framework** — a library that gives you a structure and set of rules to build an app (vs. a library you call directly).
- **Container** — a runtime component that creates, configures and manages objects for you. In Spring, that's the **ApplicationContext** (explained later).
- **Bean** — an object **managed** by the Spring container. When I say "bean", think "instance created by Spring and tracked by Spring."
- **Annotation** — metadata you put in source code with `@Something`. Spring reads many annotations at runtime to decide what to do (e.g., `@Component`).
- **Dependency** — any object/class a class needs to do its job (e.g., `Car` depends on `Engine`).
- **Dependency Injection (DI)** — the mechanism of providing a dependency to a class from the outside (rather than the class creating it).
- **Inversion of Control (IoC)** — the general principle: components don't create their dependencies; a container (Spring) creates and wires them. DI is a way to achieve IoC.
- **Proxy** — a wrapper object that stands in for another object and can intercept calls to that object (used for AOP, transactions).
- **Aspect (AOP)** — a modular unit of cross-cutting behavior (logging, transactions). We'll cover AOP later.

---

1. WHY IoC & DI matter

Problem without DI: classes `new` their dependencies, leading to tight coupling, hard-to-test code, and scattered object creation.

Benefit of DI: makes classes easy to test, replace, and configure. The container handles object lifecycle, wiring, and (often) configuration.

Analogy: Instead of building all the parts of a house yourself and wiring lights, you hire an electrician (Spring). You tell the electrician what rooms and fixtures you want, and they wire everything. Your code asks for things — it doesn't assemble them.

Key approaches to DI

- **Constructor injection** — dependencies provided via constructor parameters. (Recommended)
- **Setter injection** — dependencies provided via setter methods.
- **Field injection** — dependencies injected directly into fields (possible but not recommended for production code).

Minimal runnable example (constructor injection)

```
// GreetingService.java
public interface GreetingService {
    String greet();
}

// SimpleGreetingService.java
import org.springframework.stereotype.Component;

@Component
public class SimpleGreetingService implements GreetingService {
    @Override
    public String greet() { return "Hello from SimpleGreetingService"; }
}

// AppRunner.java
import org.springframework.stereotype.Component;

@Component
public class AppRunner {
    private final GreetingService greetingService;
    public AppRunner(GreetingService greetingService) {
        this.greetingService = greetingService;
    }
    public void run() {
        System.out.println(greetingService.greet());
    }
}

// DemoApplication.java
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ConfigurableApplicationContext;

@SpringBootApplication
public class DemoApplication {
    public static void main(String[] args) {
        try (ConfigurableApplicationContext ctx =
SpringApplication.run(DemoApplication.class, args)) {
            AppRunner runner = ctx.getBean(AppRunner.class);
            runner.run();
        }
    }
}
```

What happened: `@Component` tells Spring "make a bean of this class." Spring automatically wires `SimpleGreetingService` into `AppRunner` via its constructor.

---

2. How Spring *finds* and *registers* beans: component scanning & stereotypes

New term — Component scanning: the process where Spring examines packages for classes annotated with certain annotations (like `@Component`) and registers them as beans.

Stereotype annotations (what they are):

- `@Component` — generic component; marks a class as a bean.
- `@Service` — semantic variant for service/business layer (same behavior).
- `@Repository` — semantic variant for data-access layer; also triggers exception translation (converts persistence exceptions to Spring's `DataAccessException` hierarchy).
- `@Controller` — for web MVC controllers.
- `@RestController` — `@Controller + @ResponseBody` — for REST endpoints returning JSON.

Important rule: `@SpringBootApplication` (or `@ComponentScan`) enables component scanning for the package it's in and subpackages. Place your main class at package root.

Example — component scanning

If main class is `com.example.demo.DemoApplication`, Spring scans `com.example.demo.*` packages for `@Components`.

---

3. ApplicationContext vs BeanFactory — container basics

New term — BeanFactory: the very basic Spring container interface; it can create and manage beans lazily.

New term — ApplicationContext: a richer container that builds on BeanFactory and adds:

- Message resolution (i18n),
- Event propagation,
- Convenient bean lookup and lifecycle features,
- Support for AOP and profiles.

Practical rule: use `ApplicationContext` (you almost always will via Spring Boot).

How to get the context: `ConfigurableApplicationContext ctx = SpringApplication.run(App.class, args);` — you seldom need it, but it's what hosts the beans.

---

4. Bean lifecycle — the sequence of events (very important)

New term — Bean definition vs bean instance:

- **Bean definition** = metadata about how to create a bean (class, scope, init method).
- **Bean instance** = the actual object created.

Bean lifecycle steps (typical):

1. **Bean definition** read by container.
2. **Instantiation** — container creates the object (by calling constructor).
3. **Populate properties / resolve dependencies** — inject other beans.
4. **BeanNameAware / BeanFactoryAware / ApplicationContextAware** callbacks (if implemented).
    - *Definition:* These are Spring interfaces a bean can implement to receive references (e.g., the `ApplicationContext`) from the container.
5. **`BeanPostProcessor` pre-initialization processing** (hooks).
    - *Definition:* `BeanPostProcessor` is an interface you implement when you want to intercept bean creation (e.g., to wrap beans with proxies).
6. **Initialization callbacks** (`@PostConstruct`, `afterPropertiesSet()`, custom `initMethod`).
    - `@PostConstruct` is an annotation (Jakarta) for a method run after dependencies are injected.
7. **Bean ready for use** — your app uses the bean.
8. **Shutdown** — container calls destroy callbacks (`@PreDestroy`, `DisposableBean#destroy()`, custom `destroyMethod`).

Example — init/destroy

```
import jakarta.annotation.PostConstruct;
import jakarta.annotation.PreDestroy;
import org.springframework.stereotype.Component;

@Component
public class MyBean {
    @PostConstruct
    public void init() { System.out.println("init"); }

    @PreDestroy
    public void cleanup() { System.out.println("destroy"); }
}
```

`BeanPostProcessor` example (brief):

```
import org.springframework.beans.factory.config.BeanPostProcessor;
import org.springframework.stereotype.Component;

@Component
public class MyBeanPostProcessor implements BeanPostProcessor {
    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName) {
        // called before @PostConstruct
        return bean;
    }
    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName) {
```

```
        // called after @PostConstruct; useful to wrap bean with proxy
        return bean;
    }
}
```

---

5. Bean scopes — what they mean and pitfalls

New term — Bean scope: life-cycle/visibility policy for beans.

Common scopes:

- **singleton (default)** — single instance per Spring container. (Not the same as the singleton pattern.) Most beans are singletons.
- **prototype** — a new instance returned every time the bean is requested from the container.
- **request** — one instance per HTTP request (web apps).
- **session** — one instance per HTTP session (web apps).

Important detail (prototype lifecycle): The container instantiates and injects dependencies for prototype beans, but **it does not manage their destruction** (`@PreDestroy` is not called by container for prototypes). You must clean up manually.

How to configure scope:

```
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

@Component
@Scope("prototype")
public class TempObject { }
```

Pitfall — injecting prototype into singleton: If a singleton bean has a prototype dependency via direct injection, the prototype is created once (at injection time) and reused. Use `ObjectFactory<T>`, `Provider<T>`, or `@Lookup` methods to get a fresh prototype per use.

---

6. Wiring: how Spring resolves which bean to inject

By default: Spring autowires by **type** (the class/interface).

If multiple beans of same type exist, Spring chooses by:

1. **@Primary** — a bean marked @Primary becomes the default.
2. **@Qualifier** — you can label beans and request by qualifier.
3. **Bean name** (if no qualifier/primary) — if injection point is a field/parameter named `emailService`, Spring may look for a bean named `emailService`.

Examples

```
@Service("emailNotifier")
```

```
public class EmailNotifier implements Notifier { ... }

@Service("smsNotifier")
public class SmsNotifier implements Notifier { ... }

@Component
public class AlertService {
    // By type there are two Notifier beans -> must disambiguate
    public AlertService(@Qualifier("smsNotifier") Notifier notifier) { this.notifier =
notifier; }
}
```

@Primary example

```
@Component
@Primary
public class DefaultNotifier implements Notifier { ... }
```

---

7. Injection styles — pros & cons (detailed)

Constructor injection

- **Definition:** dependencies provided via constructor parameters.
- **Pros:** immutable fields, explicit required dependencies, easy to test, recommended.
- **Cons:** long constructors if many deps (but that indicates a design smell).

Setter injection

- **Definition:** Spring calls setter methods to inject dependencies.
- **Pros:** optional dependencies are easy, circular dependency resolution sometimes possible.
- **Cons:** not guaranteed to be set at construction time.

Field injection

- **Definition:** Spring injects into private fields (e.g., `@Autowired private Repo repo;`)
- **Pros:** simple for examples.
- **Cons:** harder to test (requires reflection), hides dependencies, not recommended for production.

Example (constructor + optional)

```
@Component
public class ServiceA {
    private final Repo repo;
    private final Optional<Cache> cache; // optional

    public ServiceA(Repo repo, Optional<Cache> cache) {
        this.repo = repo;
        this.cache = cache;
    }
}
```

---

8. Configuration classes, @Bean, and when to use them

New term — @Configuration class: a class annotated with `@Configuration` that contains `@Bean` methods to programmatically construct beans.

When to use `@Bean`:

- You need to create an instance of a third-party class (no `@Component` available).
- You need fine-grained control over creation (conditional creation, parameters).
- You want to centralize configuration.

Important detail — `proxyBeanMethods`: `@Configuration` classes by default are proxied so that when one `@Bean` method calls another, the container ensures singletons (it returns the container-managed bean, not a new instance). That prevents accidental double-instantiation.

Example

```
@Configuration
public class AppConfig {
    @Bean
    public ObjectMapper objectMapper() {
        return new ObjectMapper(); // third-party class, not @Component
    }
}
```

---

9. Externalized configuration — properties, @Value, and @ConfigurationProperties

New term — externalized configuration: keeping environment-specific values (DB URLs, credentials, feature toggles) outside code (in `application.properties` or YAML files), so you don't hard-code them.

How to provide config (Spring Boot typical):

- `src/main/resources/application.properties` or `application.yml`
- Profiles can override settings (e.g., `application-dev.properties`)

@Value (single-value injection):

```
app.greeting=Hello, user!
@Component
public class Greeter {
    @Value("${app.greeting}")
    private String greeting;
}
```

@ConfigurationProperties (grouped, type-safe binding)

- Use when you have many related properties; binds them into a POJO.

- Supports nested properties, lists, validation.

```
app.mail.host=smtp.example.com
app.mail.port=587
app.mail.username=me
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Component;

@Component
@ConfigurationProperties(prefix = "app.mail")
public class MailProperties {
    private String host;
    private int port;
    private String username;
    // getters & setters
}
```

Why prefer `@ConfigurationProperties` over many `@Value`?

- Type safety, grouping, easier testing and documentation.

Environment abstraction

- **New term — Environment:** Spring's `Environment` abstraction exposes property sources and active profiles and allows programmatic access to properties: `env.getProperty("app.greeting")`.

---

10. Profiles — separate development and production configuration

New term — profile: a named logical grouping (like `dev`, `prod`) that controls which beans and property sets are active.

How to use:

- Annotate beans: `@Profile("dev")` — then they only load when `dev` profile is active.
- Put properties in `application-dev.properties`.

How to activate profiles:

- CLI: `--spring.profiles.active=dev`
- Environment variable: `SPRING_PROFILES_ACTIVE=dev`
- In `application.properties` during dev (not recommended for production builds).

Example:

```
@Profile("dev")
@Bean
public DataSource devDataSource() { ... }

@Profile("prod")
@Bean
```

```
public DataSource prodDataSource() { ... }
```

Important: Profiles control bean creation and property selection — a lot of configuration uses profiles for safe defaults.

---

11. Conditional bean creation (advanced but useful)

New term — condition: a runtime check that decides whether a bean should be created.

Spring Core provides `@Conditional` for custom conditions. Spring Boot adds many *convenience* conditional annotations like `@ConditionalOnProperty` (Boot-specific). For core, learn `@Conditional`.

Simple example (conceptual)

```
@Conditional(MyCondition.class)
@Bean
public SomeBean someBean() { ... }
```

`MyCondition` implements `Condition` and checks environment or classpath.

---

12. AOP (Aspect-Oriented Programming) — concepts and mechanics

New term — AOP: Programming technique to separate cross-cutting concerns (logging, security, transactions) into modular aspects.

Fundamental terms:

- **Aspect** — a modular cross-cutting concern (class annotated `@Aspect`).
- **Join point** — a point during the execution of the program, such as method call/return.
- **Pointcut** — an expression that selects join points (e.g., `execution(* com.example..*(..))`).
- **Advice** — action taken at a join point (before, after, around).
- **Weaving** — the process of applying aspects to target objects (using proxies in Spring).

How Spring implements AOP: proxy-based using either:

- **JDK dynamic proxies** (when bean implements an interface), or
- **CGLIB** (subclassing) when there is no interface (or proxyTargetClass=true).

Important detail — proxy implications:

- Proxies only intercept *external* method calls through the proxy. **Self-invocation** (a method in the same class calling another method on `this`) bypasses the proxy — aspects/transactions won't be applied.
- Final classes and final methods can prevent proxying (CGLIB subclassing can't override `final` methods).

Example — logging aspect

```
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class LoggingAspect {
    @Around("execution(* com.example..*(..))")
    public Object log(ProceedingJoinPoint pjp) throws Throwable {
        long start = System.currentTimeMillis();
        Object ret = pjp.proceed();
        long time = System.currentTimeMillis() - start;
        System.out.printf("%s took %dms%n", pjp.getSignature(), time);
        return ret;
    }
}
```

When to use AOP: Use for logging, metrics, security checks, caching, retry logic — things you want separated from business logic.

---

13. Transactions — what they are and @Transactional behavior

New term — transaction: a set of operations that must succeed or fail together (atomicity). In databases, a transaction groups multiple SQL operations into one atomic unit.

New term — transaction manager: a Spring abstraction that coordinates transactions for a particular resource (e.g., `PlatformTransactionManager` for JDBC/JPA).

What `@Transactional` does:

- Starts a transaction before the method executes,
- Commits on successful completion,
- Rolls back on runtime (unchecked) exceptions by default,
- Can be configured for propagation, isolation, read-only flags.

Where to place `@Transactional`: typically on **service layer** methods (public methods). Putting it on DAOs is possible but not recommended.

Important proxy interaction:

- Spring uses proxies for `@Transactional`. If you call a `@Transactional` method from another method in the same class, the transaction proxy is bypassed (self-invocation), so no transaction will be applied.
- Solution: move transactional methods to another bean or use AspectJ weaving (advanced).

Common attributes:

- `propagation` — how transactions relate to existing ones (`REQUIRED` default, `REQUIRES_NEW`, etc.).
- `isolation` — defines read phenomena allowed (`READ_COMMITTED`, `SERIALIZABLE`, etc.).
- `readOnly` — optimization hint.
- `rollbackFor` — specify exceptions to trigger rollback.

Example

```
import org.springframework.transaction.annotation.Transactional;
import org.springframework.stereotype.Service;

@Service
public class AccountService {
    private final AccountRepository repo;
    public AccountService(AccountRepository repo) { this.repo = repo; }

    @Transactional
    public void transfer(Long fromId, Long toId, int amount) {
        Account from = repo.findById(fromId).orElseThrow();
        Account to = repo.findById(toId).orElseThrow();
        from.withdraw(amount);
        to.deposit(amount);
        repo.save(from);
        repo.save(to);
        // exception here -> rollback
    }
}
```

---

14. Spring Data (short note) & Repositories

New term — Spring Data JPA: a Spring project that builds repository implementations at runtime from interface definitions; it reduces boilerplate for JPA-based persistence.

Why learn Data JPA early: Spring Boot + Spring Data JPA is a common stack. Understanding repositories (`JpaRepository`) helps you focus on service logic instead of CRUD boilerplate.

Repository example

```
public interface SkillRepository extends JpaRepository<Skill, Long> {
    boolean existsByName(String name);
    Page<Skill> findByCategory(String category, Pageable p);
}
```

(This is Boot-related but useful; learning the repository pattern later is easy once you know DI, transactions and entities.)

---

15. Events — ApplicationEventPublisher and listeners

New term — event: a message that something happened; publisher/listener pattern.

Use-case: decouple components — publish domain events and react asynchronously/synchronously elsewhere.

Example

```
public class UserCreatedEvent {
    private final Long userId;
    public UserCreatedEvent(Long userId) { this.userId = userId; }
    public Long getUserId() { return userId; }
}

// publishing
@Autowired
private ApplicationEventPublisher publisher;
publisher.publishEvent(new UserCreatedEvent(userId));

// listening
@Component
public class UserCreatedListener {
    @EventListener
    public void onUserCreated(UserCreatedEvent ev) { /* handle */ }
}
```

---

16. Resource loading & Environment (practical utilities)

New term — Resource: an abstraction that points to a file-like object (classpath, filesystem, URL). Spring's `ResourceLoader` can load `classpath:foo.txt`, `file:/tmp/foo.txt`, `http://...`.

Environment & property sources: Spring's `Environment` exposes active profiles and property sources. Use `Environment` when you need to read properties programmatically.

---

17. Advanced bean concepts (FactoryBean, lifecycle callback interfaces)

New term — FactoryBean: a special bean that acts as a factory for other objects. Implement `FactoryBean<T>` to programmatically create complex objects and expose them as normal beans.

Example use case: when creating dynamic proxies or when object creation needs complex logic.

Lifecycle interfaces

- `InitializingBean` — implement `afterPropertiesSet()` (alternative to `@PostConstruct`).
- `DisposableBean` — implement `destroy()`.

Prefer annotations (`@PostConstruct`, `@PreDestroy`) over these interfaces for testability and decoupling.

---

18. Common pitfalls and how to avoid them (practical checklist)

1. **Self-invocation and proxies:** If a method annotated with `@Transactional` or targeted by an aspect is invoked from another method in the same class, the proxy won't intercept it. **Fix:** move the annotated method into a separate bean or use `AopContext.currentProxy()` (advanced and not ideal) or enable AspectJ weaving (complex).
2. **Final classes/methods & proxies:** proxies rely on subclassing (CGLIB) or JDK proxies. `final` prevents subclassing; avoid `final` on beans you expect to proxy.
3. **Field injection hides dependencies:** prefer constructor injection for clearer, testable code.
4. **Prototype beans and destruction:** container won't call `@PreDestroy` on prototype beans — manage cleanup manually.
5. **Circular dependencies:** Constructor injection prevents circular dependencies (the container cannot construct two beans that depend on each other via constructors). Use setter injection to allow circular references, but the design may be suspect. Better approach: refactor to break cycle.
6. **Lazy vs eager initialization:** Many beans instantiate at startup. Use `@Lazy` for expensive beans you want to delay. Boot sets lazy-init options too.
7. **Returning JPA entities directly from controllers:** With lazy-loaded relationships, you can get `LazyInitializationException`. Use DTOs or ensure proper fetch strategy and transaction boundaries.
8. **Mixing frameworks / classpath issues:** Using multiple versions of Jakarta/Javax packages can cause runtime errors. Stick to consistent Spring Boot starters for compatibility.

---

19. Small, focused hands-on project — Coffee Machine (Spring Core practice)

Goal: practice bean creation, DI, scopes, properties, lifecycle, and simple component scanning. This is intentionally tiny — it focuses on Spring Core patterns.

Project structure (package `com.example.coffee`)

```
src/
  main/
    java/com/example/coffee/
      DemoApplication.java
      machine/
         CoffeeMachine.java
         Heater.java
         HeaterImpl.java
         Pump.java
         PumpImpl.java
      config/
         CoffeeProperties.java
      runner/
         AppRunner.java
    resources/
      application.properties
```

Files

DemoApplication.java

```
package com.example.coffee;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class DemoApplication {
  public static void main(String[] args) {
    SpringApplication.run(DemoApplication.class, args);
  }
}
```

Heater.java

```
package com.example.coffee.machine;
public interface Heater {
    void on();
    void off();
    boolean isHot();
}
```

HeaterImpl.java

```
package com.example.coffee.machine;
import org.springframework.stereotype.Component;
@Component
public class HeaterImpl implements Heater {
    private boolean hot = false;
    @Override
    public void on(){ hot = true; System.out.println("Heater on"); }
    @Override
    public void off(){ hot = false; System.out.println("Heater off"); }
    @Override
    public boolean isHot(){ return hot; }
}
```

Pump.java

```
package com.example.coffee.machine;
public interface Pump {
    void pump();
}
```

PumpImpl.java

```
package com.example.coffee.machine;
import org.springframework.stereotype.Component;

@Component
public class PumpImpl implements Pump {
    @Override
    public void pump() { System.out.println("Pumping water..."); }
}
```

CoffeeMachine.java

```
package com.example.coffee.machine;
import org.springframework.stereotype.Component;
import jakarta.annotation.PostConstruct;
import jakarta.annotation.PreDestroy;
```

```
@Component
public class CoffeeMachine {
    private final Heater heater;
    private final Pump pump;

    public CoffeeMachine(Heater heater, Pump pump) {
        this.heater = heater;
        this.pump = pump;
    }

    @PostConstruct
    public void init() { System.out.println("CoffeeMachine starting..."); }

    public void brew() {
        heater.on();
        pump.pump();
        System.out.println("Brewing coffee!");
    }

    @PreDestroy
    public void stop() { System.out.println("CoffeeMachine stopping..."); heater.off(); }
}
```

`CoffeeProperties.java` (example of `@ConfigurationProperties`, Boot specific)

```
package com.example.coffee.config;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Component;

@Component
@ConfigurationProperties(prefix = "coffee")
public class CoffeeProperties {
    private String strength = "NORMAL";
    // getter & setter
    public String getStrength(){ return strength; }
    public void setStrength(String strength){ this.strength = strength; }
}
```

`AppRunner.java`

```
package com.example.coffee.runner;
import com.example.coffee.machine.CoffeeMachine;
import com.example.coffee.config.CoffeeProperties;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

@Component
public class AppRunner implements CommandLineRunner {
    private final CoffeeMachine machine;
    private final CoffeeProperties props;

    public AppRunner(CoffeeMachine machine, CoffeeProperties props) {
        this.machine = machine;
        this.props = props;
    }

    @Override
    public void run(String... args) throws Exception {
        System.out.println("Coffee strength is: " + props.getStrength());
        machine.brew();
    }
```

```
}
```

`application.properties`

```
coffee.strength=STRONG
```

How to run

- Create a Spring Boot project with `spring-boot-starter` (or `starter-web` if you prefer). Add `spring-boot-starter` and `spring-boot-configuration-processor` if you want metadata for IDEs.
- `mvn spring-boot:run` — output shows bean initialization, property binding and `brew()` output.

What you practiced

- `@Component` scanning and wiring (DI),
- constructor injection,
- `@PostConstruct`/`@PreDestroy`,
- `@ConfigurationProperties` binding,
- `CommandLineRunner` to run code at startup.