Spring Boot

Wings 1

Sep 12, 2025

Table of Contents

1. Spring Boot: Overview & Architecture

What is Spring Boot?

- A module built on top of the Spring Framework to make creating stand-alone, production-grade Spring (and often Spring MVC, Data, Security, etc.) applications with minimal configuration easier.
- Provides *opinionated defaults*, auto-configuration, embedded servers, externalized configuration, monitoring, etc.

Key architectural goals:

- *Convention over Configuration*: pick reasonable defaults, reduce boilerplate.
- *Autoconfiguration* based on classpath, beans, properties.
- Embedded server support.
- Expose production-ready features (metrics, healthchecks).
- External configuration (properties, YAML, profiles).

High-Level Layers:

Often we see these logical layers:

- Presentation / Web / API (Controllers)
- Service / Business logic
- Data Access / Repository (e.g. Spring Data)
- Domain / Model / Entities
- Infrastructure layer (external integrations, security, scheduling, etc.)
- Configuration & Bootstrapping

2. Application Startup Flow

Let’s go step by step through what happens **when you run** a Spring Boot app.

Entry Point

```
@SpringBootApplication

public class MyApp {

    public static void main(String[] args) {

        SpringApplication.run(MyApp.class, args);

    }

}
```

- `@SpringBootApplication` is a composite/meta annotation. It includes:
  - `@SpringBootConfiguration` (which is a specialization of `@Configuration`)
  - `@EnableAutoConfiguration`
  - `@ComponentScan` (with default package being the package of the class)
- `SpringApplication.run(...)` starts the Spring context, parses arguments, sets up environment, etc.

The startup phases roughly are:

1. **Bootstrap**:
   - Create `SpringApplication` instance.
   - Get sources, initial configurations.
2. **Environment preparation**:

    - Determine active profiles (`spring.profiles.active`), environment variable overrides, command-line args, default properties.

3. **ApplicationContext creation**:
    - Depending on whether it's web or non-web application, create `AnnotationConfigServletWebServerApplicationContext` or `AnnotationConfigApplicationContext` etc.
4. **Register listeners**:
    - ApplicationListeners, ApplicationContextInitializers, etc.
5. **Load bean definitions**:
    - From `@Configuration` classes, `@ComponentScan`, imported configurations, auto-configurations.
6. **Refresh context**:
    - Finish creating beans, apply post processors, etc.
7. **Start embedded web server** (if web application)
    - Create and start server (Tomcat/Jetty/Undertow).
8. **Call `CommandLineRunners` and `ApplicationRunners`** (if any)
9. **App is ready to serve requests**

A diagram:

```
[main()]

   ↓

SpringApplication.run()

   ↓

Set up Environment (Properties, Profiles)
```

```
   ↓
Create and configure ApplicationContext

   ↓
ComponentScan + AutoConfiguration → bean definitions

   ↓
BeanFactoryPostProcessors / BeanPostProcessors

   ↓
Instantiate singletons, resolve dependencies

   ↓
Embedded server started

   ↓
ApplicationRunners / CommandLineRunners

   ↓
Ready state – accept HTTP requests
```

3. Auto-Configuration Mechanism & `@EnableAutoConfiguration`

This is central to Spring Boot's "magic".

- `@EnableAutoConfiguration` (or via `@SpringBootApplication`) causes Spring Boot to search *classpath* and see what dependencies are present, what beans are already defined, what properties are set, etc., and then load matching auto-configuration classes.

- Auto-configuration classes are located in `META-INF/spring.factories` (pre Spring Boot 2.7 / 3.0) or in newer Boot versions also `spring.autoconfigure` etc. They are discovered via Spring's `SpringFactoriesLoader`.

- Each auto-configuration class has conditions (meta-annotations) like `@ConditionalOnClass`, `@ConditionalOnMissingBean`,

`@ConditionalOnProperty`, `@ConditionalOnBean`, etc. These ensure only relevant configuration kicks in.

- E.g. if `spring-boot-starter-web` is on classpath, then Tomcat (or default server) and Spring MVC auto-configs are loaded.

- If a developer defines an explicit bean of a certain type, conditional on missing bean means auto-configuration won't override.

Examples:

```
@Configuration

@ConditionalOnClass({Servlet.class, DispatcherServlet.class})

@ConditionalOnMissingBean(DispatcherServlet.class)

public class DispatcherServletAutoConfiguration { ... }
```

So Boot checks: “are we a web application? Is servlet API present? Do we already have a DispatcherServlet bean? If yes, skip; else define one.”

- Order matters: Spring Boot has `@AutoConfigureOrder` and some priorities.

4. Component Scanning & Dependency Injection

Component Scanning

- The `@ComponentScan` (implicitly included via `@SpringBootApplication`) looks in the base package (package of the class with the annotation) and its subpackages for components to register. These include:
    - `@Component`
    - `@Service`
    - `@Repository`

    - `@Controller` / `@RestController`
    - Any `@Configuration` class.
- Scanning can be customized: limit the packages, include/exclude filters, etc.

Dependency Injection (DI)

- Spring's core is the IoC container. Beans are instantiated and dependencies injected.
- Injection types:
    - Constructor injection (recommended).
    - Setter injection.
    - Field injection (less preferred but works).
- Injection via `@Autowired`, `@Inject`, or via constructor if only one constructor (since Spring 4.3+).
- Also `@Qualifier` when multiple beans of a type.
- Lifecycle order of injection etc.

5. Bean Lifecycle & Scopes

Bean Scopes

Common scopes:

Scope	Bean lifecycle / behaviour
`singleton`	One shared instance per Spring ApplicationContext (default)
`prototype`	A new instance every time it is injected / retrieved

`request`	One per HTTP request (in web context)
`session`	One per HTTP session
`application`	ServletContext scope

others in WebFlux etc.

Lifecycle hooks

- `@Bean` methods in `@Configuration` classes define beans explicitly.
- `@PostConstruct` / `@PreDestroy` on bean methods for initialization / destruction.
- InitializingBean / DisposableBean interfaces.
- `BeanPostProcessor` – before/after initialization.
- `BeanFactoryPostProcessor`, `BeanDefinitionRegistryPostProcessor` – allow moving or customizing bean definitions *before* beans are created.
- Lazy initialization: using `@Lazy` or setting default lazy behavior.
- Bean ordering: using `@DependsOn`, `@Order`, or `Ordered` interface.

6. Controller-Service-Repository Layer Architecture

Logical separation:

- **Controller** layer – handles HTTP / Web requests, mapping, validation, request/response DTOs. Annotations: `@Controller`, `@RestController`, `@RequestMapping`, `@GetMapping` etc.
- **Service** layer – business logic, data transformations, transactional boundaries. Annotation: `@Service`.

- **Repository** layer – data access. Using Spring Data, e.g. `@Repository`, interfaces that extend `JpaRepository`, `CrudRepository`, custom queries.
- **Entity / Model** layer – domain objects, JPA entities.
- **DTO / Mapper** as needed (to decouple internal model from exposed API).

Flow: Client → Controller → Service → Repository → Database → back up → Response.

7. Configuration: application.properties / application.yml

- Standard files placed in `src/main/resources`:
  - `application.properties` or `application.yml`
- Properties include:
  - Server port: `server.port=8080`
  - Data source config: `spring.datasource.url=...`, `spring.datasource.username=...` etc.
  - JPA / Hibernate – dialect, ddl-auto, show_sql etc.
  - Logging, Actuator, etc.
- YAML allows hierarchical configuration. Profiles via `application-<profile>.yml/properties`.
- Properties are loaded in certain precedence (application.properties, command-line args, environment vars etc.)

8. Externalized Configuration: `@Value`, `@ConfigurationProperties`

- `@Value("${property.name}")` on fields / constructor params to inject a specific value. Good for simple / few properties.

- `@ConfigurationProperties(prefix="myapp")` to bind a group of properties from file into a strongly typed class. Supports nested, validation via JSR-303 (using `@Validated`).
- These also work with profiles.

9. Spring Boot Starters & Dependency Setup

- Starters are just dependency descriptors that help you bring in sets of libraries. Examples:
  - `spring-boot-starter-web` – includes spring MVC, embedded Tomcat, Jackson etc.
  - `spring-boot-starter-data-jpa`, `spring-boot-starter-security`, `spring-boot-starter-actuator`, etc.
- Using starters:
  - Simplifies POM / Gradle setup.
  - Ensures version compatibility (starter versions are aligned with Boot version).

10. Embedded Server & Lifecycle

- Spring Boot includes embedded servlet containers (Tomcat by default). You can switch to Jetty, Undertow.
- During startup:
  - Auto-configuration for web server: e.g. `TomcatServletWebServerFactory` bean, which creates/configures the embedded servlet container.
  - `WebServer` is started once ApplicationContext is refreshed.
- Lifecycle events:

    - `ServletWebServerInitializedEvent` when server is ready.
    - Graceful shutdown hooks.
- If packaged as WAR and deployed in external container, things differ — SpringBootServletInitializer is used.

11. Request Flow in REST APIs

Putting it all together, request flow for a typical REST endpoint:

1. Client sends HTTP request (e.g. `GET /api/users/123`).
2. Embedded server (Tomcat etc.) accepts the request, starts a thread, has the `DispatcherServlet` (in Spring MVC) as front controller.
3. DispatcherServlet uses HandlerMapping to find which controller method matches (by annotations `@RequestMapping`, `@GetMapping` etc.).
4. The request is parsed (path variables, query params, request body deserialization etc.).
5. `@RestController` (annotated) method is invoked with dependencies injected.
6. Controller may do validation (via `@Valid`, etc.), then call service layer.
7. Service interacts with repository, domain entities, maybe transaction begins.
8. Repository interfaces (Spring Data) generate queries or use custom ones, data fetch / persist.
9. Response is built (often with DTO), serialized (e.g. via Jackson) to JSON.
10. Response returned back to client.

12. Exception Handling

- Default behavior: unhandled exceptions result in error responses (often 500). Spring provides default error handling via `BasicErrorController`.

- Customization:
    - `@ControllerAdvice` + `@ExceptionHandler` methods allow global/targeted exception handling.
    - You can define specific handlers, return custom response, status codes, etc.
    - You can also override `ErrorController`.
- Validation errors (via `@Valid`) produce `MethodArgumentNotValidException`, etc.

13. Data Access & Spring Data JPA

- `@Entity` classes map to database tables; JPA annotations (`@Table`, `@Id`, etc.).
- Repositories:
    - `CrudRepository`, `JpaRepository`, `PagingAndSortingRepository` etc.
    - Custom query methods: via method naming convention, `@Query`, native queries.
- Underneath, Hibernate (or other JPA provider) handles ORM.
- Configuration of datasource, JPA dialect, DDL auto (create, update, validate), connection pooling etc.

14. Transaction Management

- `@Transactional` annotation (on methods or classes).
- Defines boundaries: begin transaction, commit/rollback.
- Transaction management via proxies (AOP).
- Default propagation rules, isolation levels, rollback rules etc.

- For example, a `RuntimeException` triggers rollback by default; checked exceptions do not unless configured.

15. Security Basics

- Spring Security is integrated via `spring-boot-starter-security`.
- Typical setup:
    - Define a `@Configuration` class with `@EnableWebSecurity`.
    - Extend `WebSecurityConfigurerAdapter` (older versions) or configure beans of type `SecurityFilterChain` (newer versions).
    - Filters: authentication filters, authorization filters.
    - Authentication providers (in-memory, JDBC, JWT, etc.).
- Boot auto-configures default basic auth if security on classpath unless overridden.

16. Actuator & Health Checks

- `spring-boot-starter-actuator` adds endpoints for monitoring & managing application (health, metrics, metrics/prometheus, info, etc.).
- By default endpoints like `/actuator/health`, `/actuator/info`.
- You can expose or hide endpoints via config.
- Useful in production: health probes, readiness/liveness, metrics.

17. Profiles & Environment-Specific Configuration

- Use `@Profile("dev")`, `@Profile("prod")` on configuration classes, bean definitions etc.
- Use `application-dev.properties` / `application-prod.yml`, and activate via `spring.profiles.active=dev` or via environment variable or command line.

- Profiles allow different beans / settings in different environments.

18. Custom Annotations & Meta-Annotations

- Meta-annotations: e.g. define your own annotation that itself is annotated with things like `@Component` + `@Transactional` etc.
- Useful in large projects for standardizing behaviors.

Example:

```
@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Service

@Transactional

public @interface MyTransactionalService { }
```

-
- Trick: order of meta-annotations, component scanning will pick up custom ones if they include stereotypes.

19. Scheduling & Asynchronous Tasks

- `@EnableScheduling` + methods annotated `@Scheduled(fixedRate=..., cron=...)`.
- `@EnableAsync` + methods annotated `@Async`. Requires a task executor.
- These tasks run in background, outside HTTP request flow.
- Be careful with thread pools, exception handling in async methods etc.

20. Testing Spring Boot

- `@SpringBootTest` – loads full application context for integration tests.

- `@WebMvcTest` – slice test just for controller layer without full context.
- `@DataJpaTest` for repository/data layer.
- `@MockBean` to mock beans in context.
- `@TestConfiguration` for test-specific beans.
- Test with properties override, profiles for test, in-memory databases (H2 etc.)

21. Common Edge Cases & Tricky Scenarios

Some pitfalls / corner cases that often show up in interviews or can break in real use:

- **Circular dependencies**: e.g. A depends on B, B depends on A. Usually fails at startup unless using setter or field injection + `@Lazy`.
- **Lazy initialization**: can delay bean creation; useful to speed startup, but may move exception to runtime rather than startup.
- **Bean overriding**: two beans of same type / name, auto-configuration vs user bean. Boot allows enabling bean overriding but generally discouraged.
- **Multiple matching beans** → qualifier needed.
- **Transaction boundary issues**: calling a `@Transactional` method within same class (self-invocation) doesn't go through proxy so the transaction may not be applied.
- **Proxy vs target class issues**: interface vs class proxies, final methods not overridden etc.
- **Classpath conflicts**: e.g. two versions of the same library.
- **Profile / property precedence** gotchas (which property wins: command-line, environment variables, application.yml, defaults).
- **Thread safety in components / services** which are singletons by default.

22. Lifecycle Hooks & Initialization Order

- `ApplicationContextInitializer` hooks before context is refreshed.
- `ApplicationListener` for various events: `ApplicationStartingEvent`, `ApplicationEnvironmentPreparedEvent`, `ApplicationPreparedEvent`, `ContextRefreshedEvent`, `ApplicationReadyEvent`, etc.
- `CommandLineRunner` and `ApplicationRunner` execute after context ready.
- `@PostConstruct` / `@PreDestroy` in beans.
- `BeanPostProcessor` before/after bean's init methods.
- The order: bean definitions → BeanFactoryPostProcessors → BeanPostProcessors → instantiate singletons → dependency injection → post construct etc.

23. Custom Auto-Configuration & Conditional Annotations

- You can write your own auto-configuration classes, register them via `spring.factories` (or in newer versions via `spring.autoconfigure`).
- Use conditional annotations:
    - `@ConditionalOnProperty` – enable/disable config based on property.
    - `@ConditionalOnMissingBean` – only configure if no bean of given type.
    - `@ConditionalOnBean`, `@ConditionalOnClass`, `@ConditionalOnResource` etc.
- Also can control ordering with `@AutoConfigureBefore`, `@AutoConfigureAfter`.

24. DevTools, Hot Reloading & Productivity

- `spring-boot-devtools`: includes features like automatic restart, live reload, property defaults, developer tools.

- Changes to classpath triggers restart of embedded server.
- LiveReload: browser automatic refresh etc.
- Logging, enabling debug, actuator trace endpoints (if needed).
- Use of Lombok, etc., for reducing boilerplate.

25. Deployment (JAR vs WAR, Containerization etc.)

- **Executable JAR**: embed server, run via `java -jar`. Most common pattern nowadays.
- **WAR** packaging: if deploying to external servlet container (e.g. Tomcat, WildFly). For this, extend `SpringBootServletInitializer` and override `configure()`.
- Dockerization: writing Dockerfile, using multi-stage builds, managing config via environment variables, health check endpoints.
- Other: cloud native (K8s), Native Image (GraalVM) in newer Boot versions, resource usage.

MCQ-Style Notes / Commonly Tested Concepts & Possible Trick Questions

These are good to know for certification / interview MCQs:

- *What does `@SpringBootApplication` include?* → `@Configuration`, `@EnableAutoConfiguration`, `@ComponentScan`.
- *If you define your own `DataSource` bean, will auto-configuration still configure one?* → No, because many auto-configs are conditional on missing bean, so yours will take priority.
- *What happens if the application is non-web but you have `spring-boot-starter-web` on classpath?* → Boot may assume web application, embed server etc. (you can disable via `spring.main.web-application-type=none`).

- *Propagation of transactions for checked vs unchecked exceptions.*
- *Self-invocation of `@Transactional` methods doesn't work via proxy.*
- *Bean scope: do prototype beans get `@PreDestroy` called?* → No, Spring doesn't manage their full lifecycle once handed out.
- *Order of property overriding: command line > environment vars > application-properties > defaults.*
- *Activating profiles, or which profile(s) are active when none is set.*
- *LazyInitializationException in Hibernate: when accessing un-initialized proxies outside transaction or session.*
- *How filters (security, servlet filters) vs interceptors work in request flow.*
- *When `@ComponentScan` is not scanning certain packages (if base package configured incorrectly).*

Real-World Understanding: Data Flow & Layers Relationship

Here's a typical data/request flow in a Spring Boot REST microservice:

1. Client (mobile app / browser / external system) makes HTTP request.
2. Embedded server (Tomcat etc.) receives request → DispatcherServlet.
3. DispatcherServlet → resolves path via HandlerMapping → Controller method.
4. Controller:
    - Parses request body / query params.
    - Validates via `@Valid` + binding results.
    - Transforms DTO to domain model.
5. Controller calls Service:

    - Contains business logic.
    - May call multiple repositories.
    - Might call external services, do transformations, checks, apply transaction.
6. Repository layer:
    - Spring Data repository method → JPA / Hibernate executes SQL or native query.
    - Fetch or persist entities.
7. Back in Service, result (entities) may be mapped to DTOs.
8. Controller returns DTO → serialized to JSON (via Jackson or other) → HTTP response.
9. On errors / exceptions, global exception handler catches and forms proper response code & body.
10. Also, cross-cutting concerns applied: logging filters, security filters, transaction proxies, AOP advice etc.

Example Code Snippets & Key Annotations

Here are focused code pieces to illustrate certain things.

Custom Auto-Configuration Example

```
@Configuration

@ConditionalOnClass(SomeService.class)

@ConditionalOnProperty(prefix="my.feature", name="enabled",
havingValue="true", matchIfMissing=false)

public class MyFeatureAutoConfiguration {


    @Bean

    @ConditionalOnMissingBean(MyFeatureService.class)
    public MyFeatureService myFeatureService() {
        return new MyFeatureServiceImpl();
    }
}
```

And in `META-INF/spring.factories`:

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration
=\
com.example.autoconfig.MyFeatureAutoConfiguration
```

Transaction + Self‑invocation Gotcha

```
@Service
public class OrderService {

    @Transactional
    public void placeOrder(Order order) {
        // some logic
        applyDiscount(order);
        // more logic
    }

    @Transactional
    public void applyDiscount(Order order) {
        // but since this is call from same class, the proxy
won't intercept and transactional semantics may not apply
    }
}
```

Fix: split into another bean or call through the proxy.

Diagrams / Flowcharts

(Since text only, I'll describe key diagram you might draw for interview or MCQ:)

- Startup flow: from `main` → SpringApplication → Environment → ApplicationContext → BeanDefinition loading (ComponentScan + AutoConfig) → Bean instantiation (singletons) → server start → Ready.
- Request processing flow: client → server → DispatcherServlet → HandlerMapping → Controller → Service → Repository → DB → back → Response.
- Bean lifecycle: Definition -> Instantiation -> Populate Properties (DI) -> `@PostConstruct` -> bean ready / use → `@PreDestroy` on shutdown (for singleton) etc.

Example: Putting It All Together (Simple REST Service)

```
@SpringBootApplication
public class DemoApp {
    public static void main(String[] args) {
        SpringApplication.run(DemoApp.class, args);
    }
}

@RestController
@RequestMapping("/api/users")
public class UserController {
```

```java
    private final UserService userService;

    public UserController(UserService userService) {
        this.userService = userService;
    }

    @GetMapping("/{id}")
    public ResponseEntity<UserDTO> getUser(@PathVariable Long
id) {
        UserDTO dto = userService.getUserById(id);
        return ResponseEntity.ok(dto);
    }
}

@Service
public class UserService {

    private final UserRepository userRepo;

    public UserService(UserRepository userRepo) {
        this.userRepo = userRepo;
    }

    @Transactional(readOnly=true)
    public UserDTO getUserById(Long id) {
        User user = userRepo.findById(id)
             .orElseThrow(() -> new NotFoundException("User not found"));
        return mapToDTO(user);
    }
}

@Repository
public interface UserRepository extends JpaRepository<User, Long> {
}

@Entity
public class User {

    @Id @GeneratedValue
    private Long id;
    private String name;
    // getters/setters
}

@ControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(NotFoundException.class)
    public ResponseEntity<ErrorResponse>
handleNotFound(NotFoundException ex) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND)
                 .body(new ErrorResponse("NOT_FOUND",
ex.getMessage()));
    }
}
```

---

25. Deployment, Dockerization

- Build as JAR: default.
- If want WAR: change packaging, extend `SpringBootServletInitializer`, override `configure(SpringApplicationBuilder builder)`.
- Docker: write Dockerfile, expose port, use environment variables to set configuration.
- Health endpoints important for container orchestration (Kubernetes etc.).
- Consider layered jars (Boot 2.3+ “layers”) for more efficient docker builds.

---

26. MCQ / Interview‑Trick Questions Examples

Here are some sample MCQs / “which of the following is true/false” style with tricky options:

1. **Which annotation(s) does `@SpringBootApplication` include?**

   A. `@EnableAutoConfiguration`

B. `@Configuration`
C. `@ComponentScan`
D. `@EnableWebMvc`

Correct: A, B, C. *Not* `@EnableWebMvc` (unless you explicitly include it).

2. **If two beans of the same type exist, and one is marked `@Primary`, which one is injected when using `@Autowired` without qualifier?**

    ○ The one marked `@Primary`.

3. **Propagation behavior: If method A (non-transactional) calls method B (annotated `@Transactional`) in same class, is B run in a transaction?**

    ○ No, because self-invocation bypasses proxy.

4. **Which of the following are conditional annotations in Spring Boot auto-configuration?**

    A. `@ConditionalOnClass`
    B. `@ConditionalOnMissingBean`
    C. `@ConditionalOnProperty`
    D. `@ConditionalOnBean`

    → All are correct.

5. **Does `@PostConstruct` get called for prototype beans via container?**

    ○ Yes, on creation; but `@PreDestroy` is *not* called by container for prototype scoped beans.

6. **Order of property/value resolution (highest precedence first):**

    A. Command-line arguments
    B. application-<profile>.properties
    C. Environment variables
    D. application.properties inside jar

    One correct sequence: Command line > Environment variables > application-profile

> application.properties > default.

7. **If you have `spring.main.lazy-initialization=true`, when are beans instantiated?**

   - Only when first needed rather than at startup.

8. **Which of the following is *not* part of Spring Boot's auto-configuration?**

   A. DispatcherServlet
   B. Jackson JSON converter (if on classpath)
   C. Hibernate validator
   D. Custom beans in user code

   → D. Because custom beans are user-provided, not auto-configured (but they may be picked up via scanning).

---

27. Summary

Spring Boot gives you a highly modular, convention-driven, auto-configured environment for developing production ready Java applications. Understanding its flow — from startup, bean creation, dependency injection, request handling, configuration, to lifecycle hooks — is crucial both for exam/certification questions and for building robust systems in real life.

🧠 1. Core / Boot Configuration Annotations

@SpringBootApplication

- **Definition**: Main entry point for a Spring Boot application.

  Example:

```
 @SpringBootApplication
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```

- **Includes**: `@Configuration`, `@EnableAutoConfiguration`, `@ComponentScan`.
- **Use**: On main class.
- **Best Practice**: Place in a root package to ensure full scanning.

@Configuration

- **Definition**: Marks a class as a configuration class that defines beans.

  Example:

```
 @Configuration
public class AppConfig {
    @Bean
    public MyBean bean() { return new MyBean(); }
}
```

- **Use**: On Java classes that define `@Bean` methods.
- **Note**: Used in both Spring and Spring Boot.

@EnableAutoConfiguration

- **Definition**: Tells Spring Boot to auto-configure beans based on classpath and properties.

  Example:

  ```
  @EnableAutoConfiguration
  public class MyConfig { }
  ```

- **Use**: Rarely used directly; it's part of `@SpringBootApplication`.

@ComponentScan

- **Definition**: Enables component scanning for `@Component`, `@Service`, etc.

  Example:

  ```
  @ComponentScan(basePackages = "com.example.app")
  ```

- **Use**: On configuration classes.
- **Tip**: Keep your main class in the top package to scan sub-packages by default.

🧱 2. Component / Bean / Stereotype Annotations

@Component

- **Definition**: Generic stereotype for any Spring-managed bean.

  Example:

  ```
  @Component
  public class MyHelper { }
  ```

- **Use**: When other stereotypes (like `@Service`) don't apply.

@Service

- **Definition**: Specialization of `@Component` for service classes (business logic).

  Example:

  ```
   @Service
  public class UserService { }
  ```

- **Use**: On service layer classes.
- **Tip**: Helps with clarity and potential AOP features.

@Repository

- **Definition**: Marks a class as a Data Access Object (DAO).

  Example:

  ```
   @Repository
  public interface UserRepo extends JpaRepository<User,
  Long> { }
  ```

- **Use**: On persistence/repository classes.
- **Note**: Enables exception translation for DB errors.

@Controller

- **Definition**: Marks a web controller for handling HTTP requests.

  Example:

  ```
   @Controller
  public class WebController { }
  ```

- **Use**: When returning views (e.g., Thymeleaf templates).

@RestController

- **Definition**: Combines `@Controller` + `@ResponseBody`.

  Example:

  ```
   @RestController
  public class ApiController { }
  ```

- **Use**: For REST APIs that return JSON/XML.

@Bean

- **Definition**: Declares a method that returns a Spring-managed bean.

  Example:

  ```
   @Bean
  public MyService myService() {
      return new MyService();
  }
  ```

- **Use**: Inside `@Configuration` classes.

🌐 3. Web / MVC / REST Annotations

@RequestMapping

- **Definition**: Maps HTTP requests to handler methods.

  Example:

```
    @RequestMapping(value = "/users", method =
RequestMethod.GET)
```

- **Use**: On class or method.

- **Tip**: Prefer `@GetMapping`, `@PostMapping` for clarity.

@GetMapping, @PostMapping, @PutMapping, @DeleteMapping, @PatchMapping

- **Definition**: Shortcuts for `@RequestMapping(method=...)`.

  Example:

  ```
  @GetMapping("/users/{id}")
  public User getUser(@PathVariable Long id) { ... }
  ```

@RequestParam

- **Definition**: Binds query parameters to method arguments.

  Example:

  ```
  public void search(@RequestParam String name)
  ```

@PathVariable

- **Definition**: Binds URI path variable to a method parameter.

  Example:

  ```
  public User getUser(@PathVariable Long id)
  ```

@RequestBody

- **Definition**: Binds JSON/XML request body to a method parameter.

  Example:

  ```
  public void create(@RequestBody UserDto user)
  ```

@ResponseBody

- **Definition**: Returns the method result directly as HTTP response body.
- **Use**: On controller methods or via `@RestController`.

@ResponseStatus

- **Definition**: Sets HTTP response status for controller methods or exceptions.

Example:

```
@ResponseStatus(HttpStatus.CREATED)
```

@ExceptionHandler

- **Definition**: Handles exceptions thrown in controller methods.

  Example:

  ```
  @ExceptionHandler(ResourceNotFound.class)
  public ResponseEntity<?> handleNotFound() { ... }
  ```

🗃️ 4. Data / JPA / Persistence Annotations

@Entity

- **Definition**: Marks a class as a JPA entity.

  Example:

  ```
  @Entity
  public class User { }
  ```

@Id

- **Definition**: Marks the primary key field.

  Example:

  ```
  @Id
  private Long id;
  ```

@GeneratedValue

- **Definition**: Specifies generation strategy for primary key.

Example:

```
@GeneratedValue(strategy = GenerationType.IDENTITY)
```

@Column

- **Definition**: Configures column name, length, nullable, etc.

  Example:

  ```
  @Column(name = "email", nullable = false)
  ```

@OneToMany, @ManyToOne, @OneToOne, @ManyToMany

- **Definition**: Define relationships between entities.

  Example:

  ```
  @OneToMany(mappedBy = "user")
  private List<Order> orders;
  ```

@Transactional

- **Definition**: Marks method/class to run in a DB transaction.

  Example:

  ```
  @Transactional
  public void saveOrder() { ... }
  ```

@Query

- **Definition**: Defines a custom JPQL query in Spring Data JPA.

  Example:

  ```
  @Query("SELECT u FROM User u WHERE u.email = :email")
  ```

@Modifying

- **Definition**: Indicates a modifying (write) query.

Example:

```
@Modifying
@Query("UPDATE User u SET u.name = :name WHERE u.id = :id")
```

⚙️ 5. Conditional / Auto Configuration (Spring Boot)

@ConditionalOnClass

- **Definition**: Load bean/config only if a class is on classpath.

  Example:
  `@ConditionalOnClass(DataSource.class)`

@ConditionalOnMissingBean

- **Definition**: Load beans only if a certain bean is not present.
- **Use**: To override auto-configurations.

@ConditionalOnProperty

- **Definition**: Load config/bean only if property has a specific value.

  Example:
  `@ConditionalOnProperty(name="feature.enabled", havingValue="true")`

@Profile

- **Definition**: Load bean/config only under a certain Spring profile.

  Example:
  `@Profile("dev")`

🔧 6. Other Useful Annotations

@Scheduled

- **Definition**: Run method at fixed intervals or cron.

Example:

```
@Scheduled(fixedRate = 5000)
```

@EnableScheduling

- **Definition**: Enables Spring's scheduled task execution.
- **Use**: In config class.

@Async

- **Definition**: Executes a method asynchronously in a separate thread.
- **Use**: With `@EnableAsync`.