Most commonly used Spring Boot @Annotations

CORE SPRING BOOT ANNOTATIONS

@SpringBootApplication

- Entry point of a Spring Boot application.
- Combines @Configuration, @EnableAutoConfiguration, and @ComponentScan.
- Enables auto-configuration based on classpath dependencies.
- Should be placed at the root package to ensure proper component scanning.

💡**Tip:** Always place it in the root package to avoid missing component scanning.

@Configuration

- Indicates a class contains Spring bean definitions.
- Used for Java-based configuration instead of XML.
- Allows defining beans using @Bean methods.
- Ensures singleton behaviour for bean methods.

💡**Tip:** Use it for Java-based configuration classes that define beans.

CORE SPRING BOOT ANNOTATIONS

@EnableAutoConfiguration

- Enables Spring Boot's autoconfiguration mechanism based on classpath, existing beans, and properties.
- Automatically configures components like DataSource, JPA, Security, Redis, etc, without manual setup.
- Spring Boot used spring.factories for auto-configuration up to 2.6, introduced AutoConfiguration.imports in 2.7 as a transition, and fully switched to it in Spring Boot 3.0 for performance and modularity.
- Can be customized or disabled using exclude or excludeName attributes.

💡**Tip:** Auto-configuration follows *"convention over configuration"* - override only when necessary.

@ComponentScan

- Scans specified packages for Spring-managed components (@Component, @Service, @Repository, @Controller).
- Automatically registers detected classes as Spring beans.
- Default scan starts from the package of the annotated class and its sub-packages.
- Supports include/exclude filters for fine-grained control.

💡**Tip:** *Incorrect package placement* is one of the most common reasons for beans not being detected.

🔷 CORE SPRING BOOT ANNOTATIONS

@Bean

- Declares a Spring-managed bean manually.
- Commonly used for third-party or external classes.
- Allows full control over object creation and configuration.
- Lifecycle is managed by the Spring container.

💡**Tip:** Prefer when you need full control over object creation or use third-party classes.

@Component

- Marks a class as a Spring-managed component.
- Automatically detected during component scanning.
- Base stereotype for @Service, @Repository, and @Controller.
- Used for generic, reusable components.

💡**Tip:** Use for generic reusable components not tied to a specific layer.

🔷 DEPENDENCY INJECTION

@Autowired

- Automatically injects dependencies by type.
- Can be used on fields, constructors, or setters.
- Reduces manual object creation.
- Constructor injection is preferred for immutability and testing.

💡**Tip:** Prefer constructor injection for immutability and testability.

@Qualifier

- Resolves ambiguity when multiple beans of the same type exist.
- Used with @Autowired.
- Injects a specific bean by name.
- Helps avoid NoUniqueBeanDefinitionException.

💡**Tip:** Use when multiple beans of the same type exist.

@Primary

- Marks a bean as the default among multiple candidates.
- Used when @Qualifier is not specified.
- Simplifies dependency resolution.
- Common in multi-datasource setups.

💡**Tip:** Mark a default bean to avoid excessive qualifiers.

🔷 LAYERED ARCHITECTURE ANNOTATION

@Service

- Represents the business logic layer.
- Improves code readability and architectural clarity.
- Semantically indicates service-level responsibilities.
- Behaves like @Component but recommended for business logic.

💡**Tip:** Use for business logic to keep controllers thin and readable.

@Repository

- Marks data access layer (DAO).
- Enables automatic exception translation to DataAccessException.
- Used with JPA, JDBC, or other persistence frameworks.
- Improves separation of concerns.

💡**Tip:** Always use for persistence to enable exception translation.

@Controller

- Handles HTTP requests in Spring MVC.
- Typically returns views (HTML, JSP).
- Used in traditional MVC applications.
- Works with view resolvers.

💡**Tip:** Mainly use for MVC apps returning views.

@RestController

- Combines @Controller and @ResponseBody.
- Returns JSON/XML responses directly.
- Used for RESTful APIs.
- Eliminates the need to annotate each method with @ResponseBody.

💡**Tip:** Default choice for REST APIs returning JSON.

WEB & REST API ANNOTATIONS

@RequestMapping

- Maps HTTP requests to controller methods.
- Supports URL paths, HTTP methods, headers, and params.
- It can be applied at the class and method levels.
- Base annotation for all request mappings.

💡**Tip:** Use at class level for base API paths.

@GetMapping

- Specialized annotation for HTTP GET requests.
- Improves readability over @RequestMapping(method=GET).
- Used for fetching data.
- Supports path variables and query parameters.

💡**Tip:** Use for read-only, idempotent operations.

@PostMapping

- Handles HTTP POST requests.
- Used for creating resources.
- Commonly paired with @RequestBody.
- Supports validation using @Valid.

💡**Tip:** Use for resource creation with request bodies.

@PutMapping

- Handles HTTP PUT requests.
- Used for updating existing resources.
- Idempotent by design.
- Typically updates full objects.

💡**Tip:** Use for full updates of existing resources.

WEB & REST API ANNOTATIONS

@DeleteMapping

- Handles HTTP DELETE requests.
- Used to remove resources.
- Often combined with @PathVariable.
- Should return appropriate HTTP status codes.

💡**Tip:** Use only for delete operations and return proper status codes.

@PathVariable

- Extracts values from the URI path.
- Used for resource identification.
- Type-safe binding supported.
- Mandatory by default unless marked optional.

💡**Tip:** Use for identifying resources in RESTful URLs.

@RequestParam

- Extracts query parameters from URL.
- Useful for filtering and pagination.
- Supports default values and optional params.
- Throws an error if the required param is missing.

💡**Tip:** Use for filters, pagination, and optional inputs.

@RequestBody

- Converts JSON/XML request body into a Java object.
- Uses Jackson for serialization/deserialization.
- Required for POST/PUT APIs with payloads.
- Works with validation annotations.

💡**Tip:** Use for JSON payloads, never for GET requests.

@iamabhishekmaurya

WEB & REST API ANNOTATIONS

@ResponseStatus

- Sets a specific HTTP status code on the response returned by a controller method or exception.
- Can be applied at the method level or exception class level.
- Eliminates the need to wrap responses in ResponseEntity for simple cases.
- Often used to represent REST semantics like 201 Created, 404 Not Found, etc.

💡**Tip:** Use @ResponseStatus for fixed, predictable status codes, but prefer ResponseEntity when the status needs to change dynamically at runtime.

VALIDATION ANNOTATIONS

- Used to validate incoming data automatically before business logic executes.
- Based on Jakarta Bean Validation (JSR-380) and integrated seamlessly with Spring Boot.
- Triggered using @Valid or @Validated on method parameters.
- Prevents invalid data from reaching service and database layers.

@Valid

- Triggers validation on objects annotated with constraint annotations.
- Commonly used with @RequestBody, @PathVariable, or method parameters.
- Automatically throws validation exceptions when constraints fail.
- Works well for basic object-level validation.

💡**Tip:** Triggers validation automatically on request payloads.

@Validated

- Spring-specific extension of @Valid.
- Supports validation groups for conditional validation.
- Can be applied at the class or method level.
- Useful for validating method arguments in the service layer.

💡**Tip:** Use when you need validation groups or method-level validation.

🔷 VALIDATION ANNOTATIONS

@NotNull

- Ensures value is not null.
- Allows empty strings and collections.
- Suitable for mandatory fields.

💡**Tip:** Use for fields where null is unacceptable, but empty value is allowed.

@NotBlank

- Ensures string is not null and contains non-whitespace text.
- Trims whitespace before validation.
- Best for user input fields.

💡**Tip:** Prefer this for required text fields like name or title.

@NotEmpty

- Ensures a string, collection, or array is not empty.
- Allows whitespace-only strings.
- Useful for collections.

💡**Tip:** Use for lists and maps that must contain at least one element.

@Size

- Restricts the length of strings or size of collections.
- Supports min and max values.
- Prevents overly large payloads.

💡**Tip:** Always combine with @NotNull or @NotBlank if required.

🔷 VALIDATION ANNOTATIONS

@Email

- Validates email format.
- Allows customization using regex.
- Useful for contact information.

💡**Tip:** Combine with @NotBlank to avoid empty emails passing validation.

@Min / @Max

- Enforces numeric range constraints.
- Applied to numeric fields.
- Prevents invalid numeric values.

💡**Tip:** Use for IDs, age, quantity, or pagination limits.

@Pattern

- Validates string against a regular expression.
- Useful for custom formats (codes, IDs).
- Highly flexible but easy to misuse.

💡**Tip:** Keep regex simple and well-documented to avoid maintenance issues.

🔷 EXCEPTION HANDLING ANNOTATIONS

@ExceptionHandler

- Handles specific exceptions within a controller or controller advice.
- Eliminates repetitive try-catch blocks in controller methods.
- Can return custom error responses or ResponseEntity.
- Method-level scope unless used inside @ControllerAdvice.

💡**Tip:** Use @ExceptionHandler inside @RestControllerAdvice instead of controllers to keep controllers clean.

@ControllerAdvice

- Provides global exception handling across all controllers.
- Supports handling exceptions, binding errors, and model attributes.
- Improves maintainability by centralizing error logic.
- Can be limited to specific packages or annotations.

💡**Tip:** Always scope large applications using basePackages to avoid unintended exception capture.

@RestControllerAdvice

- Combines @ControllerAdvice and @ResponseBody.
- Automatically returns JSON/XML responses.
- Ideal for REST APIs.
- Ensures consistent error response format.

💡**Tip:** Use this for REST APIs to enforce a standard error contract.

JPA / DATABASE ANNOTATIONS

@Entity

- Marks a class as a JPA-managed entity.
- Maps the class to a database table.
- Must have a no-argument constructor.
- Lifecycle is managed by Hibernate.

💡**Tip:** Use entities only for persistence, never expose them directly in APIs.

@Table

- Defines table name and database-level constraints.
- Allows specifying indexes and unique constraints.
- Helps align entity naming with DB conventions.
- Optional but recommended for clarity.

💡**Tip:** Always define indexes at table level for frequently queried columns.

@Id

- Marks the primary key of the entity.
- Required for all JPA entities.
- Used to uniquely identify rows.
- Works with @GeneratedValue.

💡**Tip:** Avoid business keys as primary keys, use surrogate IDs instead.

@GeneratedValue

- Defines primary key generation strategy.
- Common strategies: IDENTITY, SEQUENCE, AUTO.
- Delegates ID generation to DB or Hibernate.
- Impacts insert performance.

💡**Tip:** Prefer SEQUENCE for high-throughput systems; avoid IDENTITY when batching.

JPA / DATABASE ANNOTATIONS

@Column

- Customizes column mapping.
- Controls nullability, length, and uniqueness.
- Useful for schema clarity.
- Optional but powerful.

💡**Tip:** Always explicitly define constraints for critical columns.

@OneToOne

- Maps one-to-one relationship between entities.
- Can be unidirectional or bidirectional.
- Uses foreign key or shared primary key.
- Supports cascading.

💡**Tip:** Avoid unless truly required, often leads to unnecessary joins.

@OneToMany

- Represents a parent-to-child relationship.
- Usually mapped by a child entity.
- Lazy-loaded by default.
- Often combined with @JoinColumn or mappedBy.

💡**Tip:** Avoid bidirectional @OneToMany unless necessary, it’s costly.

@ManyToOne

- Maps a many-to-one relationship.
- Most commonly used association.
- Eager-loaded by default.
- Defines the owning side of the relationship.

💡**Tip:** Always set fetch = FetchType.LAZY to avoid N+1 issues.

JPA / DATABASE ANNOTATIONS

@ManyToMany

- Maps many-to-many relationships.
- Uses a join table internally.
- It can quickly become complex.
- Harder to maintain.

💡**Tip:** Replace with two @ManyToOne mappings and a join entity.

@JoinColumn

- Specifies a foreign key column.
- Controls column name and constraints.
- Defines the owning side of the relationship.
- Avoids unnecessary join tables.

💡**Tip:** Always name foreign key columns explicitly.

@JoinTable

- Defines a join table for many-to-many relations.
- Allows customization of the join table name and columns.
- Used for association tables.
- Improves schema clarity.

💡**Tip:** Prefer explicit join entities over @ManyToMany.

@Enumerated

- Maps enum fields to DB columns.
- Supports ORDINAL and STRING.
- Controls how enums are stored.
- Affects schema readability.

💡**Tip:** Always use EnumType.STRING to avoid breaking changes.

JPA / DATABASE ANNOTATIONS

@Temporal

- Maps java.util.Date to DB date/time.
- Supports DATE, TIME, TIMESTAMP.
- Legacy annotation.
- Not required for java.time classes.

💡**Tip:** Use LocalDate, LocalDateTime, Instant instead of Date.

@Embedded

- Embeds a value-type object into an entity.
- Avoids creating a separate table.
- Used for reusable fields.
- Promotes clean design.

💡**Tip:** Use for audit or address-like objects.

@Embeddable

- Marks a class as embeddable.
- Used with @Embedded.
- No identity of its own.
- Stored in the same table.

💡**Tip:** Perfect for grouping related columns logically.

@MappedSuperclass

- Shares common fields across entities.
- Not mapped to a table.
- Parent fields inherited by child entities.
- Useful for base entity design.

💡**Tip:** Use for audit fields like createdAt, updatedAt.

🔷 JPA / DATABASE ANNOTATIONS

@Version

- Enables optimistic locking.
- Prevents lost updates.
- Automatically incremented.
- Ensures data consistency.

💡**Tip:** Always use optimistic locking in concurrent systems.

@Transactional

- Manages database transactions.
- Ensures atomicity and rollback.
- Supports propagation and isolation.
- Declarative transaction management.

💡**Tip:** Place at service layer, never in repositories or controllers.

🔷 CONFIGURATION & PROFILES ANNOTATIONS

@Configuration

- Marks a class as a source of bean definitions.
- Used for Java-based configuration instead of XML.
- Ensures singleton behaviour for @Bean methods via CGLIB proxying.
- Plays a key role in auto-configuration and custom configurations.

💡**Tip:** Use @Configuration for application-wide setup, not for business logic.

@Bean

- Declares a Spring-managed bean explicitly.
- Used when you cannot annotate the class directly.
- Supports custom initialization and destruction callbacks.
- Lifecycle is fully managed by the Spring container.

💡**Tip:** Prefer @Bean over @Component when configuring third-party libraries.

@Value

- Injects property values from configuration files.
- Supports default values and SpEL expressions.
- Simple and lightweight for individual values.
- Evaluated at runtime during bean creation.

💡**Tip:** Avoid using @Value excessively—maintenance becomes difficult as configs grow.

@ConfigurationProperties

- Binds external configuration to a typed POJO.
- Supports hierarchical and complex configuration structures.
- Type-safe and easy to validate.
- Encouraged replacement for multiple @Value usages.

💡**Tip:** Always use this for grouped or complex configurations like DB, Redis, or feature flags.

🔷 CONFIGURATION & PROFILES ANNOTATIONS

@EnableConfigurationProperties

- Enables support for @ConfigurationProperties classes.
- Automatically registers property-binding beans.
- Required when the config class is not annotated with @Component.
- Common in libraries and auto-configuration modules.

💡**Tip:** Use this when writing custom starters or shared configuration modules.

@PropertySource

- Loads external property files into the Spring environment.
- Useful for custom or non-standard property locations.
- Works alongside application.properties.
- Supports multiple property sources.

💡**Tip:** Avoid overusing it; prefer Spring Boot's default config loading whenever possible.

@Profile

- Activates beans only for specific environments.
- Common profiles: dev, test, prod.
- Prevents environment-specific beans from loading accidentally.
- Works with classes, methods, and @Beans.

💡**Tip:** Use profiles for infrastructure differences, not business logic branching.

@ActiveProfiles (Testing)

- Activates specific profiles during tests.
- Overrides default or system profiles.
- Useful for loading test-specific configurations.
- Works with @SpringBootTest.

💡**Tip:** Always define a test profile to isolate test configurations cleanly.

ASYNC, SCHEDULING & CACHING

@EnableAsync

- Enables asynchronous method execution.
- Used with @Async.
- Improves performance for non-blocking tasks.
- Requires task executor configuration.

💡**Tip:** Enables background execution of non-blocking tasks.

@Async

- Executes the method in a separate thread.
- Used for background tasks.
- Does not block the main thread.
- Should not be used inside the same class calls.

💡**Tip:** Use for fire-and-forget operations like emails.

@EnableScheduling

- Enables scheduled task execution.
- Required for @Scheduled.
- Used for cron jobs and fixed delays.
- Runs in background threads.

💡**Tip:** Required to run scheduled jobs.

@Scheduled

- Executes the method at fixed intervals.
- Supports cron expressions.
- Used for batch jobs and cleanup tasks.
- Should be lightweight operations.

💡**Tip:** Use for cron jobs and periodic background tasks.

ASYNC, SCHEDULING & CACHING

@EnableCaching

- Enables Spring cache abstraction.
- Required for caching annotations.
- Works with Redis, Caffeine, EhCache.
- Improves performance significantly.

💡**Tip:** Activates Spring's caching abstraction.

@Cacheable

- Caches the method result.
- Avoids repeated database calls.
- Cache key generated automatically or custom.
- Best for read-heavy operations.

💡**Tip:** Cache expensive read operations.

@CacheEvict

- Removes data from cache.
- Used after update/delete operations.
- Prevents stale data.
- Supports conditional eviction.

💡**Tip:** Clear cache after update or delete operations.

TESTING ANNOTATION

@SpringBootTest

- Loads full application context.
- Used for integration testing.
- Slower but realistic tests.
- Supports multiple environments.

💡**Tip:** Use for full integration testing.

@WebMvcTest

- Loads only the controller layer.
- Faster than full context.
- Used for API testing.
- Requires mocking service beans.

💡**Tip:** Ideal for fast controller-layer tests.

@MockBean

- Replaces real bean with mock.
- Used in Spring test context.
- Supports Mockito.
- Improves test isolation.

💡**Tip:** Replace real beans with mocks in Spring tests.

Do you think this was helpful?
Follow me for more

@iamabhishekmaurya