**5. @EnableAutoConfiguration**

**Purpose:**

* Enables **Spring Boot's auto-configuration** mechanism.
* Automatically configures beans based on classpath dependencies.

**Example:**

@Configuration

@EnableAutoConfiguration

public class MyConfig {

}

**10. @Configuration**

**Purpose:**

* Marks a class as a source of bean definitions.
* Used to define Spring beans explicitly using @Bean.

**Key Properties & Values:**

* No specific properties.

**Example:**

@Configuration

public class AppConfig {

@Bean

public MyService myService() {

return new MyService();

}

}

**1. @ComponentScan**

**Purpose:**

* Specifies the packages to scan for Spring components (@Component, @Service, @Repository, @Controller).
* Helps in automatic bean discovery.

**Key Properties & Values:**

| **Property** | **Description** | **Default Value** |
| --- | --- | --- |
| basePackages | Specifies the packages to scan for components. | "default package" |
| basePackageClasses | Scans the package of the specified class. | N/A |
| includeFilters | Specifies filters for components to include. | N/A |
| excludeFilters | Specifies filters for components to exclude. | N/A |

**Example:**

@Configuration

@ComponentScan(basePackages = "com.example.service")

public class AppConfig {

}

**4. @SpringBootApplication**

**Purpose:**

* A **combination of three annotations**:
  + @Configuration (marks it as a configuration class)
  + @EnableAutoConfiguration (enables auto-configuration)
  + @ComponentScan (scans for Spring components)
* Entry point for Spring Boot applications.

**Example:**

@SpringBootApplication

public class MyApplication {

public static void main(String[] args) {

SpringApplication.run(MyApplication.class, args);

}

}

**1. @Component**

**Purpose:**

* A generic stereotype annotation for any Spring-managed component (bean).
* Marks a class as a Spring bean, allowing it to be autodetected during classpath scanning.

**Key Properties & Values:**

* No specific properties; works as a marker.

**Example:**

@Component

public class MyComponent {

public void display() {

System.out.println("Hello from MyComponent!");

}

}

**2. @Service**

**Purpose:**

* A specialization of @Component indicating that a class provides business logic.
* Helps in differentiating service layer components.

**Key Properties & Values:**

* No specific properties; works as a marker.

**Example:**

@Service

public class MyService {

public String getMessage() {

return "Service Layer Message";

}

}

**3. @Repository**

**Purpose:**

* A specialization of @Component indicating a DAO (Data Access Object) that interacts with the database.
* Enables exception translation for persistence-related exceptions.

**Key Properties & Values:**

* No specific properties; works as a marker.

**Example:**

@Repository

public class MyRepository {

public List<String> getData() {

return Arrays.asList("Data1", "Data2", "Data3");

}

}

**4. @Controller**

**Purpose:**

* A specialization of @Component, used in MVC architecture to define web controllers.
* Works with @RequestMapping to handle HTTP requests.

**Key Properties & Values:**

* No specific properties; works as a marker.

**Example:**

@Controller

public class MyController {

@GetMapping("/welcome")

public String welcome(Model model) {

model.addAttribute("message", "Hello from Controller!");

return "welcome"; // Returns the view name

}

}

**5. @RestController**

**Purpose:**

* A combination of @Controller and @ResponseBody, used for RESTful web services.
* Every method inside it returns data directly as a response (JSON/XML).

**Key Properties & Values:**

* No specific properties; works as a marker.

**Example:**

@RestController

@RequestMapping("/api")

public class MyRestController {

@GetMapping("/greet")

public String greet() {

return "Hello from RestController!";

}

}

**6. @Autowired**

**Purpose:**

* Used for dependency injection in Spring.
* Automatically resolves dependencies without needing explicit bean configuration.

**Key Properties & Values:**

* required: true by default (throws an exception if the bean is missing).

**Example:**

@Component

public class MyClass {

private final MyService myService;

@Autowired

public MyClass(MyService myService) {

this.myService = myService;

}

public void execute() {

System.out.println(myService.getMessage());

}

}

**7. @Qualifier**

**Purpose:**

* Used alongside @Autowired to specify which bean to inject when multiple beans of the same type exist.

**Key Properties & Values:**

* value: The name of the bean to be injected.

**Example:**

@Component

public class MyClass {

private final MyService myService;

@Autowired

public MyClass(@Qualifier("specificService") MyService myService) {

this.myService = myService;

}

public void execute() {

System.out.println(myService.getMessage());

}

}

**8. @Primary**

**Purpose:**

* Used when multiple beans of the same type exist to give one of them higher preference during autowiring.

**Key Properties & Values:**

* No properties.

**Example:**

@Service

@Primary

public class PrimaryService implements MyService {

@Override

public String getMessage() {

return "Primary Service Message";

}

}

**9. @Bean**

**Purpose:**

* Used to define and configure a Spring bean explicitly in a @Configuration class.

**Key Properties & Values:**

* name: The name of the bean.

**Example:**

@Configuration

public class AppConfig {

@Bean

public MyService myService() {

return new MyService();

}

}

**1. @PostConstruct**

**Purpose:**

* Marks a method to be executed **after the bean's dependency injection is completed**.
* Typically used for initialization logic.

**Example:**

@Component

public class MyBean {

@PostConstruct

public void init() {

System.out.println("Bean has been initialized.");

}

}

**2. @PreDestroy**

**Purpose:**

* Marks a method to be executed **just before the bean is destroyed**.
* Used for cleanup activities such as closing resources.

**Example:**

@Component

public class MyBean {

@PreDestroy

public void cleanup() {

System.out.println("Bean is being destroyed.");

}

}

**6. @Transactional**

**Purpose:**

* Manages database transactions **automatically**.
* Ensures data consistency by rolling back transactions in case of failure.

**Key Properties & Values:**

| **Property** | **Description** | **Default Value** |
| --- | --- | --- |
| propagation | Defines transaction propagation behavior. | REQUIRED |
| isolation | Defines isolation level. | DEFAULT |
| readOnly | Indicates whether the transaction is read-only. | false |

**Example:**

@Service

public class MyService {

@Transactional

public void performTransaction() {

System.out.println("Transaction in progress...");

}

}

**1. @RequestMapping**

**Purpose:**

* Maps HTTP requests to handler methods in a **Spring MVC Controller**.
* Can be used at the class level (for setting a base path) or method level (for specific request mappings).

**Key Properties & Values:**

| **Property** | **Description** | **Default Value** |
| --- | --- | --- |
| value / path | Defines the URL path(s) to map. | "" |
| method | Specifies HTTP method (GET, POST, etc.). | ANY |
| params | Filters requests based on parameters. | ANY |
| headers | Filters requests based on headers. | ANY |

**Example:**

@RestController

@RequestMapping("/api")

public class MyController {

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

public String sayHello() {

return "Hello, World!";

}

}

**2. @GetMapping**

**Purpose:**

* **Shortcut for @RequestMapping(method = RequestMethod.GET)**.
* Used for handling **HTTP GET** requests.

**Example:**

@RestController

@RequestMapping("/users")

public class UserController {

@GetMapping("/{id}")

public String getUser(@PathVariable int id) {

return "User ID: " + id;

}

}

**3. @PostMapping**

**Purpose:**

* **Shortcut for @RequestMapping(method = RequestMethod.POST)**.
* Used for handling **HTTP POST** requests (typically for creating resources).

**Example:**

@RestController

@RequestMapping("/users")

public class UserController {

@PostMapping("/add")

public String addUser(@RequestBody User user) {

return "User added: " + user.getName();

}

}

**4. @PutMapping**

**Purpose:**

* **Shortcut for @RequestMapping(method = RequestMethod.PUT)**.
* Used for handling **HTTP PUT** requests (typically for updating resources).

**Example:**

@RestController

@RequestMapping("/users")

public class UserController {

@PutMapping("/{id}")

public String updateUser(@PathVariable int id, @RequestBody User user) {

return "User ID " + id + " updated to: " + user.getName();

}

}

**5. @DeleteMapping**

**Purpose:**

* **Shortcut for @RequestMapping(method = RequestMethod.DELETE)**.
* Used for handling **HTTP DELETE** requests (typically for deleting resources).

**Example:**

@RestController

@RequestMapping("/users")

public class UserController {

@DeleteMapping("/{id}")

public String deleteUser(@PathVariable int id) {

return "User ID " + id + " deleted";

}

}

**6. @PatchMapping**

**Purpose:**

* **Shortcut for @RequestMapping(method = RequestMethod.PATCH)**.
* Used for handling **partial updates**.

**Example:**

@RestController

@RequestMapping("/users")

public class UserController {

@PatchMapping("/{id}")

public String updateUserPartial(@PathVariable int id, @RequestBody User user) {

return "User ID " + id + " partially updated with new details.";

}

}

**7. @RequestBody**

**Purpose:**

* Binds the **request body** (JSON or XML) to a **Java object**.
* Used in **POST, PUT, PATCH** requests.

**Example:**

@RestController

@RequestMapping("/users")

public class UserController {

@PostMapping("/add")

public String addUser(@RequestBody User user) {

return "User added: " + user.getName();

}

}

**Request (JSON):**

json

{

"name": "John Doe",

"email": "john@example.com"

}

**8. @ResponseBody**

**Purpose:**

* Converts a method’s return **Java object** into a **JSON or XML** response.
* Typically used in **REST APIs**.

**Example:**

@RestController

@RequestMapping("/users")

public class UserController {

@GetMapping("/{id}")

@ResponseBody

public User getUser(@PathVariable int id) {

return new User(id, "John Doe", "john@example.com");

}

}

**Response (JSON):**

json

{

"id": 1,

"name": "John Doe",

"email": "john@example.com"

}

**9. @RequestParam**

**Purpose:**

* Extracts query parameters from the **request URL**.

**Key Properties & Values:**

| **Property** | **Description** | **Default Value** |
| --- | --- | --- |
| value | Parameter name. | N/A |
| required | Whether the parameter is required. | true |
| defaultValue | Default value if parameter is missing. | "" |

**Example:**

@RestController

@RequestMapping("/search")

public class SearchController {

@GetMapping("/users")

public String searchUser(@RequestParam String name, @RequestParam(defaultValue = "10") int limit) {

return "Searching for user: " + name + ", Limit: " + limit;

}

}

**Request URL Example:**

pgsql

GET /search/users?name=John&limit=5

**10. @PathVariable**

**Purpose:**

* Extracts values from **URL path parameters**.

**Key Properties & Values:**

| **Property** | **Description** | **Default Value** |
| --- | --- | --- |
| value | The path variable name. | N/A |
| required | Whether the variable is required. | true |

**Example:**

@RestController

@RequestMapping("/users")

public class UserController {

@GetMapping("/{id}")

public String getUser(@PathVariable int id) {

return "User ID: " + id;

}

}

**Request URL Example:**

bash

GET /users/123

**1. @CrossOrigin**

**Purpose:**

* Enables **Cross-Origin Resource Sharing (CORS)**.
* Used to allow requests from different origins (e.g., front-end running on a different domain).

**Key Properties & Values:**

| **Property** | **Description** | **Default Value** |
| --- | --- | --- |
| origins | Specifies allowed origins. | \* (all origins) |
| methods | Specifies allowed HTTP methods. | GET, POST, etc. |
| maxAge | Cache duration for preflight requests. | 30 min |

**Example:**

@RestController

@RequestMapping("/api")

@CrossOrigin(origins = "http://example.com") // Only allows requests from this origin

public class MyController {

@GetMapping("/data")

public String getData() {

return "CORS enabled data";

}

}

**5. @Entity**

**Purpose:**

* Marks a class as a **JPA entity** (mapped to a database table).

**Example:**

@Entity

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

}

**6. @Id**

**Purpose:**

* Marks a field as the **Primary Key** in a JPA entity.

**Example:**

@Entity

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

}

**7. @GeneratedValue**

**Purpose:**

* Defines **how the primary key is generated** automatically.

**Key Strategies:**

| **Strategy** | **Description** |
| --- | --- |
| IDENTITY | Uses database-generated auto-incremented values. |
| SEQUENCE | Uses a database sequence. |
| TABLE | Uses a table to generate IDs. |
| AUTO | Chooses strategy based on database dialect. |

**Example:**

@Entity

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

}

**8. @Column**

**Purpose:**

* Specifies **column properties** in the database.

**Key Properties:**

| **Property** | **Description** | **Default** |
| --- | --- | --- |
| name | Column name in the database. | Field name |
| nullable | Allows NULL values. | true |
| unique | Enforces unique values. | false |

**Example:**

@Entity

public class User {

@Column(name = "user\_name", nullable = false, unique = true)

private String name;

}

**9. @Table**

**Purpose:**

* Specifies **table name and properties** for an entity.

**Example:**

@Entity

@Table(name = "users")

public class User {

}

**10. @OneToMany**

**Purpose:**

* Defines a **one-to-many** relationship (one parent → multiple children).

**Example:**

@Entity

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@OneToMany(mappedBy = "user") // One user can have many orders

private List<Order> orders;

}

@Entity

public class Order {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@ManyToOne

@JoinColumn(name = "user\_id")

private User user;

}

**11. @ManyToOne**

**Purpose:**

* Defines a **many-to-one** relationship (multiple children → one parent).

**Example:**

*(See @OneToMany example above.)*

**12. @ManyToMany**

**Purpose:**

* Defines a **many-to-many** relationship (e.g., multiple students in multiple courses).

**Example:**

@Entity

public class Student {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@ManyToMany

@JoinTable(name = "student\_course",

joinColumns = @JoinColumn(name = "student\_id"),

inverseJoinColumns = @JoinColumn(name = "course\_id"))

private Set<Course> courses;

}

@Entity

public class Course {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@ManyToMany(mappedBy = "courses")

private Set<Student> students;

}

**1. @JoinColumn**

**Purpose:**

* Specifies the **foreign key column** in a database relationship.
* Used in **@OneToOne** and **@ManyToOne** relationships.

**Key Properties:**

| **Property** | **Description** | **Default Value** |
| --- | --- | --- |
| name | Defines the column name for the foreign key. | Derived from the field name |
| nullable | Determines if the column can be NULL. | true |
| unique | Enforces uniqueness constraint. | false |
| referencedColumnName | The column in the referenced entity (usually id). | Primary key |

**Example:**

@Entity

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@ManyToOne

@JoinColumn(name = "department\_id", referencedColumnName = "id")

private Department department;

}

@Entity

public class Department {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

}

*(The Employee entity has a foreign key column department\_id referencing Department.)*

**2. @Embedded**

**Purpose:**

* Embeds an **object inside another entity** (without creating a separate table).

**Example:**

@Entity

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@Embedded

private Address address;

}

@Embeddable

public class Address {

private String city;

private String country;

}

*(The Employee table will have columns for city and country.)*

**3. @Embeddable**

**Purpose:**

* Marks a class as **embeddable**, meaning it will be used inside another entity.

**Example:**

(See @Embedded example above.)

**1. @NamedQuery**

**Purpose:**

* Defines a **JPQL (Java Persistence Query Language) query** with a name that can be reused.

**Key Properties:**

| **Property** | **Description** |
| --- | --- |
| name | Unique identifier for the query. |
| query | JPQL query to execute. |

**Example:**

@Entity

@NamedQuery(name = "User.findByName", query = "SELECT u FROM User u WHERE u.name = :name")

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

}

@PersistenceContext

private EntityManager entityManager;

public User getUserByName(String name) {

return entityManager.createNamedQuery("User.findByName", User.class)

.setParameter("name", name)

.getSingleResult();

}

**2. @NamedNativeQuery**

**Purpose:**

* Defines a **native SQL query** with a name that can be reused.

**Key Properties:**

| **Property** | **Description** |
| --- | --- |
| name | Unique identifier for the query. |
| query | Native SQL query. |
| resultClass | The entity class returned by the query. |

**Example:**

@Entity

@NamedNativeQuery(name = "User.findByEmail",

query = "SELECT \* FROM users WHERE email = ?",

resultClass = User.class)

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String email;

}

**3. @Query**

**Purpose:**

* Defines a **custom JPQL or native query** inside a Spring Data JPA repository.

**Key Properties:**

| **Property** | **Description** |
| --- | --- |
| value | JPQL or SQL query. |
| nativeQuery | true if using native SQL (default: false). |

**Example:**

public interface UserRepository extends JpaRepository<User, Long> {

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

User findByEmail(@Param("email") String email);

@Query(value = "SELECT \* FROM users WHERE email = ?", nativeQuery = true)

User findByEmailNative(String email);

}

**1. @RequestHeader**

**Purpose:**

* Used to get a **specific HTTP header** from a request in Spring MVC.

**Example:**

@RestController

public class HeaderController {

@GetMapping("/getHeader")

public String getHeader(@RequestHeader("User-Agent") String userAgent) {

return "User-Agent: " + userAgent;

}

}

📌 **If the header is missing**, we can use defaultValue:

@GetMapping("/getHeader")

public String getHeader(@RequestHeader(value = "Custom-Header", defaultValue = "default-value") String headerValue) {

return "Header Value: " + headerValue;

}

**2. @CookieValue**

**Purpose:**

* Retrieves a **cookie value** from an HTTP request.

**Example:**

@RestController

public class CookieController {

@GetMapping("/getCookie")

public String getCookie(@CookieValue("sessionId") String sessionId) {

return "Session ID: " + sessionId;

}

}

📌 **If the cookie is missing**, we can use defaultValue:

@GetMapping("/getCookie")

public String getCookie(@CookieValue(value = "user", defaultValue = "Guest") String user) {

return "User: " + user;

}

**3. @SessionAttribute**

**Purpose:**

* Retrieves **session attributes**.

**Example:**

@RestController

public class SessionController {

@GetMapping("/setSession")

public String setSession(HttpSession session) {

session.setAttribute("username", "JohnDoe");

return "Session set!";

}

@GetMapping("/getSession")

public String getSession(@SessionAttribute("username") String username) {

return "Username: " + username;

}

}

**4. @ModelAttribute**

**Purpose:**

* Binds **form data** or model attributes to a method parameter.

**Example:**

@RestController

public class ModelAttributeController {

@PostMapping("/submitForm")

public String handleForm(@ModelAttribute User user) {

return "Received user: " + user.getName();

}

}

public class User {

private String name;

private int age;

// Getters and setters

}

**Jackson JSON Annotations**

These annotations control how **Spring Boot serializes/deserializes Java objects to/from JSON**.

**1. @JsonIgnore**

**Purpose:**

* Ignores a **field** during serialization (not included in JSON response).

**Example:**

public class User {

private String name;

@JsonIgnore

private String password;

}

**JSON Output:**

json

{

"name": "JohnDoe"

}

(*The password field is ignored.*)

**2. @JsonProperty**

**Purpose:**

* Maps a **JSON property name** to a different Java field name.

**Example:**

public class User {

@JsonProperty("full\_name")

private String name;

}

**JSON Input:**

json

{

"full\_name": "JohnDoe"

}

**Java Object:**  
✅ name will be set to "JohnDoe".

**3. @JsonFormat**

**Purpose:**

* Formats **dates and times** in JSON.

**Example:**

public class Event {

@JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "yyyy-MM-dd HH:mm:ss")

private Date eventDate;

}

**JSON Output:**

json

{

"eventDate": "2025-02-11 10:00:00"

}

**Spring Database Annotations**

These annotations handle **SQL scripts, transactions, and testing setup**.

**1. @Sql**

**Purpose:**

* Runs **SQL scripts** before or after a test.

**Example:**

@Sql("/data.sql") // Runs before test

public class UserRepositoryTest {

}

**2. @Rollback**

**Purpose:**

* Rolls back a transaction **after a test completes**.

**Example:**

@Test

@Rollback(true) // Rollback after test

public void testInsertUser() {

userRepository.save(new User("John"));

}

**3. @Commit**

**Purpose:**

* Commits a transaction **after a test**.

**Example:**

@Test

@Commit // Commits transaction instead of rollback

public void testInsertUser() {

userRepository.save(new User("John"));

}

**3. @Scope**

**Purpose:**

* Defines the scope of a Spring bean (singleton, prototype, etc.).

**Key Properties & Values:**

| **Property** | **Description** | **Default Value** |
| --- | --- | --- |
| value | Scope of the bean (singleton, prototype, etc.). | "singleton" |

**Example:**

@Component

@Scope("prototype")

public class MyPrototypeBean {

}

**4. @Lazy**

**Purpose:**

* Delays bean initialization until it is first requested.

**Key Properties & Values:**

| **Property** | **Description** | **Default Value** |
| --- | --- | --- |
| value | Whether to initialize lazily (true or false). | false |

**Example:**

@Component

@Lazy

public class LazyBean {

public LazyBean() {

System.out.println("LazyBean Initialized");

}

}

**6. @Value**

**Purpose:**

* Injects values from properties files or environment variables into beans.

**Key Properties & Values:**

| **Property** | **Description** | **Default Value** |
| --- | --- | --- |
| value | Expression to inject a value. | N/A |

**Example:**

@Component

public class ConfigReader {

@Value("${app.name}")

private String appName;

public String getAppName() {

return appName;

}

}

**application.properties**

app.name=My Spring Application

**3. @Lookup**

**Purpose:**

* Used to **inject a prototype-scoped bean into a singleton bean**.
* Allows dynamic retrieval of a bean instance at runtime.

**Example:**

@Component

public abstract class SingletonBean {

@Lookup

public abstract PrototypeBean getPrototypeBean();

}

@Component

@Scope("prototype")

public class PrototypeBean {

}

**7. @EnableWebSecurity**

**Purpose:**

* Enables **Spring Security** configuration.
* Allows defining security rules for a Spring Boot application.

**Example:**

@Configuration

@EnableWebSecurity

public class SecurityConfig {

}

**5. @ConfigurationProperties**

**Purpose:**

* Binds **external properties** (from application.properties or application.yml) to a class.

**Example:**

@Component

@ConfigurationProperties(prefix = "app")

public class AppProperties {

private String name;

private int version;

// Getters and Setters

}

**application.properties:**

properties

app.name=MyApplication

app.version=1

**6. @RestControllerAdvice**

**Purpose:**

* Global **exception handler** for REST controllers.

**Example:**

@RestControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(Exception.class)

public String handleException(Exception e) {

return "Error: " + e.getMessage();

}

}

**AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA**

**2. @Import**

**Purpose:**

* Allows importing other configuration classes into the current @Configuration class.
* Helps in modularizing configuration.

**Key Properties & Values:**

| **Property** | **Description** | **Default Value** |
| --- | --- | --- |
| value | The classes to import. | N/A |

**Example:**

@Configuration

@Import({ServiceConfig.class, RepositoryConfig.class})

public class AppConfig {

}

**5. @DependsOn**

**Purpose:**

* Specifies that a bean should be initialized after the specified beans.

**Key Properties & Values:**

| **Property** | **Description** | **Default Value** |
| --- | --- | --- |
| value | The names of beans this bean depends on. | N/A |

**Example:**

@Component

@DependsOn("anotherBean")

public class DependentBean {

public DependentBean() {

System.out.println("DependentBean Initialized");

}

}

**7. @PropertySource**

**Purpose:**

* Specifies an external property file to be used in Spring.

**Key Properties & Values:**

| **Property** | **Description** | **Default Value** |
| --- | --- | --- |
| value | The path to the properties file. | N/A |
| encoding | Encoding format. | "UTF-8" |

**Example:**

@Configuration

@PropertySource("classpath:app.properties")

public class AppConfig {

}

**10. @Order**

**Purpose:**

* Specifies the order of execution for Spring components or aspect processing.

**Key Properties & Values:**

| **Property** | **Description** | **Default Value** |
| --- | --- | --- |
| value | Order of execution (lower value = higher priority). | Integer.MAX\_VALUE |

**Example:**

@Component

@Order(1)

public class FirstBean {

public FirstBean() {

System.out.println("FirstBean Initialized");

}

}

@Component

@Order(2)

public class SecondBean {

public SecondBean() {

System.out.println("SecondBean Initialized");

}

}

**1. @ExceptionHandler**

**📌 Purpose:**

* Handles **specific exceptions** thrown by controller methods.
* Used within a @Controller, @RestController, or @ControllerAdvice.

**✅ Example:**

@RestController

public class DemoController {

@GetMapping("/error")

public String generateError() {

throw new IllegalArgumentException("Invalid input!");

}

@ExceptionHandler(IllegalArgumentException.class)

public String handleIllegalArgument(IllegalArgumentException ex) {

return "Error: " + ex.getMessage();

}

}

**📌 Output:**

text

Error: Invalid input!

**2. @ResponseStatus**

**📌 Purpose:**

* Sets the **HTTP response status** for a method or exception.

**✅ Example 1 (On Exception Class):**

@ResponseStatus(value = HttpStatus.BAD\_REQUEST, reason = "Invalid Request Data")

public class CustomException extends RuntimeException {

}

📌 When this exception is thrown, the response **status** will be 400 BAD REQUEST.

**✅ Example 2 (On Exception Handler Method):**

@ExceptionHandler(CustomException.class)

@ResponseStatus(HttpStatus.BAD\_REQUEST)

public String handleCustomException(CustomException ex) {

return "Handled Custom Exception!";

}

**3. @ControllerAdvice**

**📌 Purpose:**

* **Global exception handling** across multiple controllers.
* Works with @ExceptionHandler.

**✅ Example:**

@ControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(IllegalArgumentException.class)

public String handleIllegalArgument(IllegalArgumentException ex) {

return "Global Handler: " + ex.getMessage();

}

}

📌 This handles **IllegalArgumentException** **globally** for all controllers.

**4. @RestControllerAdvice**

**📌 Purpose:**

* Same as @ControllerAdvice, but specifically for **REST APIs** (@ResponseBody is included automatically).

**✅ Example:**

@RestControllerAdvice

public class GlobalRestExceptionHandler {

@ExceptionHandler(NullPointerException.class)

@ResponseStatus(HttpStatus.INTERNAL\_SERVER\_ERROR)

public String handleNullPointerException(NullPointerException ex) {

return "Something went wrong: " + ex.getMessage();

}

}

📌 This ensures **all REST API controllers** return proper JSON error responses.

**8. @Profile**

**Purpose:**

* Activates a specific Spring bean configuration based on the active profile.

**Key Properties & Values:**

| **Property** | **Description** | **Default Value** |
| --- | --- | --- |
| value | The profile name. | N/A |

**Example:**

@Component

@Profile("dev")

public class DevConfig {

public DevConfig() {

System.out.println("Dev Profile Active");

}

}

**Setting Active Profile:**

ini

-Dspring.profiles.active=dev

**9. @Conditional**

**Purpose:**

* Registers a bean conditionally based on a custom condition.

**Key Properties & Values:**

| **Property** | **Description** | **Default Value** |
| --- | --- | --- |
| value | The condition class. | N/A |

**Example:**

@Component

@Conditional(MyCondition.class)

public class ConditionalBean {

public ConditionalBean() {

System.out.println("ConditionalBean Initialized");

}

}

**Custom Condition Class:**

public class MyCondition implements Condition {

@Override

public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {

return System.getProperty("enableBean") != null;

}

}

**8. @EnableGlobalMethodSecurity**

**Purpose:**

* Enables method-level security annotations such as @PreAuthorize and @Secured.

**Key Properties & Values:**

| **Property** | **Description** | **Default Value** |
| --- | --- | --- |
| securedEnabled | Enables @Secured annotation. | false |
| prePostEnabled | Enables @PreAuthorize and @PostAuthorize. | false |

**Example:**

@Configuration

@EnableGlobalMethodSecurity(prePostEnabled = true)

public class MethodSecurityConfig {

}

@Service

public class SecureService {

@PreAuthorize("hasRole('ADMIN')")

public void adminMethod() {

System.out.println("Admin access granted.");

}

}

**9. @EnableOAuth2Sso**

**Purpose:**

* Enables **OAuth2 Single Sign-On (SSO)** in a Spring Boot application.
* Allows users to authenticate via external providers (Google, Facebook, etc.).

**Example:**

@Configuration

@EnableOAuth2Sso

public class OAuth2Config {

}

**10. @EnableJpaRepositories**

**Purpose:**

* Enables Spring Data JPA repositories.
* Required for **Spring Data JPA** to detect repository interfaces.

**Key Properties & Values:**

| **Property** | **Description** | **Default Value** |
| --- | --- | --- |
| basePackages | Package to scan for repositories. | "default package" |

**Example:**

@Configuration

@EnableJpaRepositories(basePackages = "com.example.repository")

public class JpaConfig {

}

public interface UserRepository extends JpaRepository<User, Long> {

}

**11. @EnableMongoRepositories**

**Purpose:**

* Enables **Spring Data MongoDB repositories**.
* Required when using **MongoDB** with Spring Data.

**Key Properties & Values:**

| **Property** | **Description** | **Default Value** |
| --- | --- | --- |
| basePackages | Package to scan for repositories. | "default package" |

**Example:**

@Configuration

@EnableMongoRepositories(basePackages = "com.example.repository")

public class MongoConfig {

}

public interface UserRepository extends MongoRepository<User, String> {

}

**2. @JsonIgnore**

**Purpose:**

* Excludes a field from being **serialized (sent in response)** or **deserialized (read from request)**.

**Example:**

public class User {

private String name;

@JsonIgnore

private String password; // This field will not be sent in JSON response

// Getters and setters

}

**JSON Response:**

json

{

"name": "John Doe"

}

*(Password field is ignored.)*

**3. @JsonProperty**

**Purpose:**

* Maps **JSON property name** to a **Java field**.
* Useful when JSON property name differs from Java field name.

**Example:**

public class User {

@JsonProperty("full\_name")

private String name;

}

**JSON Request:**

json

{

"full\_name": "John Doe"

}

*(Maps full\_name in JSON to name in Java.)*

**4. @JsonFormat**

**Purpose:**

* Specifies **date/time format** for JSON serialization and deserialization.

**Example:**

public class User {

@JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "yyyy-MM-dd HH:mm:ss")

private Date birthDate;

}

**JSON Response:**

json

{

"birthDate": "2025-02-11 14:30:00"

}

**4. @MappedSuperclass**

**Purpose:**

* Defines a **base class** whose fields are inherited by subclasses, but it is **not a table itself**.

**Example:**

@MappedSuperclass

public class Person {

private String name;

private String email;

}

@Entity

public class Employee extends Person {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

}

*(Table Employee will have name, email, and id, but there is no Person table.)*

**5. @Inheritance**

**Purpose:**

* Defines the **inheritance strategy** for an entity hierarchy.

**Key Strategies:**

| **Strategy** | **Description** |
| --- | --- |
| SINGLE\_TABLE | All classes stored in **one table** with a discriminator column. |
| JOINED | Each class has its **own table**, using JOIN queries. |
| TABLE\_PER\_CLASS | Each class has its **own independent table**. |

**Example:**

@Entity

@Inheritance(strategy = InheritanceType.SINGLE\_TABLE)

@DiscriminatorColumn(name = "type")

public class Animal {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

}

@Entity

@DiscriminatorValue("DOG")

public class Dog extends Animal {

private String breed;

}

*(Animals and dogs are stored in a* ***single table*** *with a type column.)*

**6. @DiscriminatorColumn**

**Purpose:**

* Used with @Inheritance(strategy = SINGLE\_TABLE).
* Specifies the **column that identifies the subclass type**.

**Example:**

(See @Inheritance example above.)

**7. @DiscriminatorValue**

**Purpose:**

* Defines the **value stored in the discriminator column** for a subclass.

**Example:**

(See @Inheritance example above.)

**8. @Version**

**Purpose:**

* Implements **optimistic locking** by maintaining a version number.

**Example:**

@Entity

public class Product {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@Version

private int version;

}

*(Each time a Product is updated, the version increases, preventing conflicts.)*

**9. @Transient**

**Purpose:**

* Marks a field **not to be persisted in the database**.

**Example:**

@Entity

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@Transient

private int calculatedAge; // Not stored in the database

}

**4. @EnableAspectJAutoProxy**

**Purpose:**

* Enables **AspectJ-based AOP** in Spring applications.

**Example:**

@Configuration

@EnableAspectJAutoProxy

public class AppConfig {

}

*(Allows Spring to process classes annotated with @Aspect.)*

**7. @EnableScheduling**

**Purpose:**

* Enables **scheduled tasks** in a Spring application.

**Example:**

@Configuration

@EnableScheduling

public class SchedulerConfig {

}

**8. @Scheduled**

**Purpose:**

* Runs a method **at fixed intervals**.

**Key Properties:**

| **Property** | **Description** |
| --- | --- |
| fixedRate | Runs at a fixed interval (in ms). |
| fixedDelay | Runs after a fixed delay (in ms). |
| cron | Uses a **cron expression** for scheduling. |

**Example:**

@Component

public class ScheduledTask {

@Scheduled(fixedRate = 5000) // Runs every 5 seconds

public void executeTask() {

System.out.println("Executing scheduled task...");

}

}

**Spring Asynchronous Execution**

**9. @EnableAsync**

**Purpose:**

* Enables **asynchronous processing** in Spring.

**Example:**

@Configuration

@EnableAsync

public class AsyncConfig {

}

**10. @Async**

**Purpose:**

* Runs a method **asynchronously** in a separate thread.

**Example:**

@Service

public class AsyncService {

@Async

public void processTask() {

System.out.println("Running in a separate thread...");

}

}

**11. @EnableCaching**

**Purpose:**

* Enables **Spring's caching mechanism**.

**Example:**

@Configuration

@EnableCaching

public class CacheConfig {

}

**12. @Cacheable**

**Purpose:**

* Caches the **result of a method**.

**Example:**

@Service

public class ProductService {

@Cacheable("products")

public Product getProductById(Long id) {

return productRepository.findById(id).orElse(null);

}

}

**13. @CachePut**

**Purpose:**

* Updates the cache **without skipping method execution**.

**Example:**

@CachePut(value = "products", key = "#product.id")

public Product updateProduct(Product product) {

return productRepository.save(product);

}

**14. @CacheEvict**

**Purpose:**

* **Removes an entry** from the cache.

**Example:**

@CacheEvict(value = "products", key = "#id")

public void deleteProduct(Long id) {

productRepository.deleteById(id);

}

**15. @EnableTransactionManagement**

**Purpose:**

* Enables **Spring’s transaction management**.

**Example:**

@Configuration

@EnableTransactionManagement

public class TransactionConfig {

}

**4. @BeforeEach**

**Purpose:**

* Runs **before each test method**.

**Example:**

public class TestExample {

@BeforeEach

void setup() {

System.out.println("Running before each test");

}

}

**5. @AfterEach**

**Purpose:**

* Runs **after each test method**.

**Example:**

@AfterEach

void cleanup() {

System.out.println("Running after each test");

}

**6. @BeforeAll**

**Purpose:**

* Runs **once before all tests** (must be static).

**Example:**

@BeforeAll

static void init() {

System.out.println("Runs once before all tests");

}

**7. @AfterAll**

**Purpose:**

* Runs **once after all tests** (must be static).

**Example:**

@AfterAll

static void tearDown() {

System.out.println("Runs once after all tests");

}

**5. @Valid**

**📌 Purpose:**

* Used to **validate objects** using Java Bean Validation (JSR-303).
* Works with @RequestBody and @ModelAttribute.

**✅ Example:**

public class UserDTO {

@NotNull

private String name;

@Min(18)

private int age;

}

@RestController

public class UserController {

@PostMapping("/addUser")

public String addUser(@Valid @RequestBody UserDTO user) {

return "User added: " + user.getName();

}

}

📌 If **validation fails**, Spring returns a **400 Bad Request** with an error message.

**6. @Validated**

**📌 Purpose:**

* Similar to @Valid, but **used at the class level**.
* Enables **group-based validation**.

**✅ Example:**

@Validated

@RestController

public class ProductController {

@GetMapping("/validatePrice")

public String validatePrice(@Min(10) @RequestParam int price) {

return "Valid price: " + price;

}

}

📌 If price is less than 10, Spring returns a **400 Bad Request**.