**Core Annotations**

1. **@SpringBootApplication**

* It is a convenience annotation that combines three key Spring annotations:
* **@SpringBootConfiguration:** It is a specialization of @Configuration. Marks the class as a source of bean definitions for the Spring IoC container.
* **@EnableAutoConfiguration:** Triggers the auto-configuration of beans based on classpath, properties, and other configurations.
* **@ComponentScan:** Automatically scans the package of the application class (and sub-packages) for Spring components. It registers classes annotated with: @Component, @Service, @Repository, @Controller, @RestController.

1. **@Configuration**

* It is a class-level annotation that indicates the class contains one or more @Bean methods and may be processed by the Spring container to generate bean definitions and service requests at runtime.

**Bean Configurations:**

* **Initialization strategy:**

**Eager** Initialization: By default, singleton beans are eagerly initialized at application startup.

**Lazy** Initialization: You can delay bean creation until it's actually needed (requested for the first time).

* **Scope:** Define how and when a bean is created and how long it lives in the container.

**Singleton** (Default): Only one instance of the bean is created per Spring container.

**Prototype**: A new instance is created every time the bean is requested.

**Request**: One bean per HTTP request

**Session**: One bean per HTTP session

**Application**: One bean per ServletContext (web app-wide singleton)

**Websocket**: One bean per WebSocket session

Example:

@Component

@Scope("singleton") // Optional since it's default

@Lazy

public class MySingletonBean {}

@Bean

@Scope("singleton")

@Lazy

public MySingletonBean mySingletonBean() {

return new MySingletonBean();

}

1. **@Bean**

* @Bean is a method-level annotation that tells Spring to invoke the annotated method and register its return value as a bean in the Spring application context.
* Unlike @Component, which relies on component scanning, @Bean is used within a class annotated with @Configuration to manually define beans and their dependencies, where @Configuration class is itself a Spring-managed bean.
* Use @Bean
* When third-party libraries/classes cannot be annotated with @Component.
* To have fine-grained control over the bean creation (constructor parameters, method logic, etc.)

**Dependency Injection & Lifecycle**

1. **@Autowired**

* The @Autowired annotation in Spring is used for Dependency Injection (DI). It allows Spring to resolve and inject collaborating beans (dependencies) into your bean automatically.
* Spring uses Reflection and the ApplicationContext to resolve the appropriate bean.
* If more than one bean of the same type are available in the ApplicationContext, the framework will throw NoUniqueBeanDefinitionException. To resolve this conflict, we can use the @Qualifier(“beanName”) annotation to specify explicitly which bean we want to inject.
* Or we can use @Primary, to specify Spring to make use of this bean by default if there's a conflict.

**Field Injection:**

@Autowired

private PaymentService paymentService;

**Field Injection is discouraged. Why?**

Field-injected dependencies are mutable. You can’t mark them as final, which makes the object state less safe.

Field injection tightly couples your class to Spring. With constructor injection, your class can be instantiated manually with plain Java, which is more testable and portable.

**Constructor Injection:** Preferred way as it encourages immutability, Better for unit testing (can use constructor-based mocking).

private final PaymentService paymentService;

@Autowired

public OrderService(PaymentService paymentService) {

this.paymentService = paymentService;

}

**Setter Injection:**

private PaymentService paymentService;

@Autowired

public void setPaymentService(PaymentService paymentService) {

this.paymentService = paymentService;

}

**Optional Autowiring:** If the bean isn’t available, Spring will not fail.

@Autowired(required = false)

private DiscountService discountService;

In traditional XML configuration (before annotations were mainstream), Spring supported different autowire modes:

byName Injects dependency by property name.

byType Injects by data type. If more than one bean of the type exists, error occurs.

Spring Boot’s component scanning and auto-configuration make XML config obsolete for 99% of applications.

**When Spring encounters multiple beans of the same type, it will throw a NoUniqueBeanDefinitionException. To avoid this you can use either @Qualifier or @Primary annotation.**

1. **@Qualifier**

* @Qualifier is used to tell Spring exactly which bean to inject among the multiple beans available of the same type.
* It works in conjunction with @Autowired.
* By default, the bean name is the class name with the first letter in lowercase. If you want to change the name, use @Component("myCat")

@Component

public class Dog implements Animal {

public String speak() {return "Woof";}

}

@Component

public class Cat implements Animal {

public String speak() {return "Meow";}

}

@Component

public class AnimalService {

@Autowired

@Qualifier("cat")

private Animal animal;

}

1. **@Primary**

* It is a Spring annotation that marks one bean as the default choice when multiple candidates of the same type exist in the application context.
* You can still override @Primary using @Qualifier.

@Component

@Primary

public class Car implements Vehicle {

public String start() { return "Car started"; }

}

1. **@Value**

* Used for injecting property values from application.properties.
* For multiple properties, prefer @ConfigurationProperties over scattered @Value.

@Value("${app.version:1.0}") // If app.version is not found, 1.0 will be used.

private String appName;

@Value("${app.supported-languages}") // Injecting Lists or Arrays

private List<String> languages;

@ConfigurationProperties(prefix = "app")

public class AppProperties {

private String name;

private String version;

}

1. **@PostConstruct**

* It is used on a method that needs to be executed after the bean is fully initialized, i.e., afterdependency injection is complete. To initialize a bean after all its dependencies are set.
* Useful for setup tasks that depend on injected fields (e.g., opening a connection, validating configurations, etc.).
* Lifecycle Sequence: Constructor ➡ Dependencies injected ➡ @PostConstruct

Example: This init() method is automatically called after the constructor and after Spring sets up all dependencies. The method must not have parameters.

@Component

public class EmailService {

@PostConstruct

public void init() {

System.out.println("EmailService initialized with host: " + smtpHost);

}

private final String smtpHost = "smtp.example.com";

}

1. **@PreDestroy**

* It used on a method that should be executed before a bean is destroyed by the Spring container.
* It is used to perform cleanup before the bean is removed.
* Spring internally uses a BeanPostProcessor to detect and call these annotations. So they’re a part of the Spring bean lifecycle management system.

Example: This method will be called just before the Spring container shuts down. The method must have no arguments.

@Component

public class EmailService {

@PreDestroy

public void cleanup() {

System.out.println("Cleaning up EmailService...");

// Close connection, release resources, etc.

}

}

1. **@Lazy**

* @Lazy tells Spring not to create a bean at startup, but to wait until it's actually needed.
* By default, Spring creates singleton beans eagerly at application startup. But sometimes, you may not want to initialize a bean unless it's actually used.

You can apply @Lazy in two ways:

1. On a Bean/Component: This class won’t be instantiated until it's injected or explicitly retrieved.

@Component

@Lazy

public class HeavyService {

public HeavyService() {

System.out.println("HeavyService bean initialized");

}

}

2. On an Injection Point: Even if HeavyService is a regular (non-lazy) bean, adding @Lazy on the injection will defer its creation until it’s first used in the method.

@Component

public class MyService {

@Autowired

@Lazy

private HeavyService heavyService;

public void useHeavyService() {

heavyService.doSomething();

}

}

1. **@Scope**

* @Scope is a Spring annotation that defines the lifecycle and visibility of a Spring bean — i.e., how many instances of a bean are created and how long they live.
* Common Bean Scopes in Spring: singleton, prototype, request, session, application, websocket.

Example:

@Component

@Scope("prototype")

public class MyBean {

...

}

@Bean

@Scope("singleton")

public MyBean myBean() {

return new MyBean();

}

**Component Annotations**

1. **@Component**

* It is a stereotype annotation, part of Spring’s component scanning mechanism, allowing Spring to automatically detect and register beans (i.e., objects managed by the Spring IoC container).
* When Spring Boot starts up, it scans the classpath for classes annotated with @Component and registers them as beans in the application context. Once registered, you can inject it anywhere using @Autowired.
* Spring scans for @Component by default in the same package as the main class (annotated with @SpringBootApplication) and its sub-packages. To customize scanning add: @ComponentScan(basePackages = "com.example.other") on the main class just after @SpringBootApplication annotation.
* It is the foundation for more specific annotations like @Service and @Repository

@Service, @Repository, and @Component are stereotype annotations used to define Spring-managed components, allowing Spring to detect and register these classes during component scanning.

1. **@Service**

* Specialized form of @Component used to annotate service layer classes. Used for classes that holds business logic.
* If you're using Aspect-Oriented Programming (AOP), it makes sense to annotate business logic classes with @Service, so cross-cutting concerns like logging, security, or transactions are clearly scoped.

1. **@** **Repository**

* Specialized form of @Component used to annotate data access layer (DAO) classes. Used for classes that directly interact with the database.
* Automatically converts exceptions thrown by JPA/Hibernate (like SQLException, JPAException) into Spring’s unified DataAccessException hierarchy.

**Annotations for Building REST APIs**

**@Controller and @ResponseBody are used to define how a class or method handles HTTP requests and how responses are returned.**

1. **@Controller**

* Class level annotation.
* Works as specialization of @Component annotation. Marks the Java class as a Spring MVC controller.
* Used to define a web request handler.
* Typically returns view names (like JSPs, Thymeleaf templates).

1. **@ResponseBody**

* Both method and class level.
* It is used to indicate that the return value of a method should be written directly to the HTTP response body in the format specified by the content type. Spring uses Accept header in HTTP requests to determine the response format.
* The conversion of the return value to the appropriate format (e.g., JSON or XML) is handled by 'HttpMessageConverters'. By default, Spring Boot includes Jackson as the JSON converter and JAXB for XML.
* In Spring Boot, @ResponseBody, by default, converts return type data into JSON format for all non-string return types (e.g. Employee, etc.). For String, @Responsebody will not do any conversion of returning data.
* Methods annotated with @ResponseBody do not use view resolution mechanisms, meaning they do not resolve or return a view name, unlike traditional Spring MVC methods that return a view name and use a view resolver.
* You can annotate the entire controller class with @ResponseBody if you want all methods to behave the same way.

1. **@RestController**

* Class level annotation.
* It is a convenience annotation that combines @Controller and @ResponseBody.
* All methods in a class annotated with @RestController will have their return values written directly to the response body.

1. **@RequestMapping**

* Class and Method level annotation. However at method level we have more modern/handy annotations available.
* At the class level, it acts as a base URI for all methods inside. At the method level, it maps a specific HTTP method and URI.
* It's used to define a request URI path and map it to a controller method or class.
* Attributes: name, path, value, method, params, headers, consumes, produces.
* @GetMapping, @PostMapping, @PutMapping, @PatchMapping, @DeleteMapping are specific variants of @RequestMapping annotation.

In Spring MVC (typically used in Spring Boot applications), annotations like @RequestBody, @RequestParam, @PathVariable, @RequestHeader, and @ModelAttribute are used to bind incoming HTTP request data to method parameters in controller classes.

1. **@RequestBody**

* Method Parameter level annotation.
* It is used to bind the body of an HTTP request to a method parameter in a controller.
* It tells Spring to deserialize the request body into a Java object using an HttpMessageConverter. By default, Spring Boot uses Jackson to convert JSON data in the request body to a Java object.
* The Content-Type header of the HTTP request determines which HttpMessageConverter is used. For example, application/json triggers the JSON converter.
* When combined with the @Valid annotation, @RequestBody can trigger validation on the bound object, leveraging Bean Validation.

@PostMapping("/createUser")

public ResponseEntity<String> createUser(@RequestBody User user) { ... }

1. **@RequestParam**

* Binds a query parameter, form data, or URL parameter to a method parameter.
* Query params come after ? in the URL. Good for search filters, pagination, sorting, etc.

Uri: GET /greet?name=Alice

@GetMapping("/greet")

public String greetUser(@RequestParam String name) { ... }

1. **@PathVariable**

* Binds a URI template variable to a method parameter. Value is embedded in the URI itself.

Uri: GET /user/123

@GetMapping("/user/{id}")

public String getUser(@PathVariable("id") Long userId) { ... }

1. **@RequestHeader**

* Binds a HTTP header value to a method parameter.

@GetMapping("/check")

public String checkHeader(@RequestHeader("User-Agent") String userAgent) { ... }

1. **@ModelAttribute**

* Binds form data or query parameters to a model object. Useful for binding multiple fields into an object (esp. in forms).

Body: name=Alice&email=alice@example.com //in case of a POST req

Uri: GET /register?name=Alice&email=alice@example.com //in case of a GET req

@PostMapping("/register")

public String registerUser(@ModelAttribute User user) { ... }

1. **@ResponseStatus**

* It is used to mark a method or an exception class with the status code and reason that should be returned in the HTTP response.
* If you throw an exception annotated with @ResponseStatus, it bypasses the need for @ExceptionHandler unless you want to customize the body.

@ResponseStatus(value = HttpStatus.NOT\_FOUND, reason = "User Not Found")

public class UserNotFoundException extends RuntimeException {

public UserNotFoundException(String message) {

super(message);

}

}

@GetMapping("/users/{id}")

public User getUser(@PathVariable Long id) {

return userRepository.findById(id)

.orElseThrow(() -> new UserNotFoundException("User with ID " + id + " not found"));

}

**Data and Validation Annotations**

1. **@Entity**

* Marks a Java class as a JPA entity, meaning it’s mapped to a database table. Each instance of this class corresponds to a row in the table.
* Every @Entity class must have a public or protected no‑argument constructor so that the persistence provider can instantiate it via reflection.
* Must define exactly one @Id (or embedded id) field.
* Entity classes cannot be final, and their persistent fields must not be static or transient (unless annotated @Transient).
* Example: @Entity(name = "User")

1. **@Table**

* Overrides default table mapping (which would otherwise use the entity’s class name).
* Attributes: name, schema, uniqueConstraints.
* Example: @Table(name = "users", schema = "public", uniqueConstraints = {@UniqueConstraint(columnNames = {"email"})})

1. **@Id, @GeneratedValue and @SequenceGenerator**

* @Id marks a field as the primary key.
* @GeneratedValue configures how the PK is generated.
* Attributes for GeneratedValue: strategy, generator. Strategies: AUTO, IDENTITY, SEQUENCE, TABLE.
* AUTO: Lets the JPA provider pick the “best” generation strategy.
* IDENTITY: Relies on a databases built‑in auto‑increment. The database generates the key when the row is inserted.
* SEQUENCE: Uses a dedicated database sequence object. You can tune how many values Hibernate fetches in one go, so you can batch many inserts in one round‑trip.
* TABLE: Simulates a sequence by storing and incrementing a counter in a dedicated table. Used on legacy DBs without auto‑increment or sequences.
* Example:

@Id

@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "user\_seq")

@SequenceGenerator(name = "user\_seq", sequenceName = "user\_sequence", allocationSize = 10)

1. **@Column**

* Customizes the column for a field.
* Attributes: name, nullable, unique, length, insertable, updateable, columnDefinition.

1. **@Temporal**

* Used for java.util.Date and Calendar to specify the temporal type. With new date-time APIs (Java8) this annatation is not required.
* Example: @Temporal(TemporalType.TIMESTAMP)

1. **@** **Transient**

* Marks a field not to be persisted in the DB. Useful for calculated or helper fields.

1. **@Enumerated**

* Maps an enum to a column.
* Attributes: EnumType STRING or ORDINAL. Used to specify whether the enum should be persisted by name or by ordinal (default)
* Example: @Enumerated(EnumType.STRING)

1. **@Embedded and @Embeddable with @AttributeOverrides and @AttributeOverride**

* Marks a class whose instances will be embedded in an entity. Embeds an @Embeddable object.
* Example:

@Embeddable

public class ContactPerson {

private String firstName;

private String lastName;

// standard getters, setters

}

@Entity

public class Company {

@Id

@GeneratedValue

private Integer id;

@Embedded

@AttributeOverrides({

@AttributeOverride( name = "firstName", column = @Column(name = "contact\_first\_name")),

@AttributeOverride( name = "lastName", column = @Column(name = "contact\_last\_name"))

})

private ContactPerson contactPerson;

// standard getters, setters

}

1. **@OneToOne**

//class User, Owning side

@OneToOne(cascade = CascadeType.ALL)

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

private Address address;

//class Address, Owing side

@OneToOne(mappedBy = "address")

private User user;

* Place the @JoinColumn annotation to configure the name of the column in the users table that maps to the primary key in the address table.
* Whoever owns the foreign key column gets the @JoinColumn annotation.
* MappedBy is required on the owing side.
* Default fetch is eager, so beware N+1.

1. **@OneToMany and @ManyToOne**

//class Email

@ManyToOne(fetch = FetchType.LAZY)

@JoinColumn(name = "employee\_id")

private Employee employee;

//class Employee

@OneToMany(fetch = FetchType.LAZY, mappedBy = "employee")

private List<Email> emails;

* The owning side is usually defined on the many side of the relationship using @JoinColumn annotation. It’s usually the side that owns the foreign key.
* To make this association bidirectional, we can easily use the mappedBy attribute of @OneToMany annotation to do so.
* In situations when we want to create multiple join columns, we can use the @JoinColumns annotation.
* Default fetch type is LAZY.
* Use orphanRemoval=true to auto‑delete child entities when removed from the collection.

@JoinColumns({

@JoinColumn(name="ADDR\_ID", referencedColumnName="ID"),

@JoinColumn(name="ADDR\_ZIP", referencedColumnName="ZIP")

})

1. **@ManyToMany**

//class Student

@ManyToMany

@JoinTable(

name = "course\_like",

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

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

Set<Course> likedCourses;

//class Course

@ManyToMany(mappedBy = "likedCourses")

Set<Student> likes;

* A many-to-many relationship doesn’t have an owner side in the database, we could configure the join table in both the tables.
* If neither side uses mappedBy, you must define a @JoinTable.

1. **@Access**

* Both class and member level.
* Used to determine whether JPA uses an entity’s fields or its getter and setter methods to interact with the database
* The @Access annotation has to be used with AccessType property. AccessType.FIELD or AccessType.PROPERTY.
* By default, the access type is defined by the place where we put the identifier annotation (@Id). If we put it on the field – it will be AccessType.FIELD, if we put it on the getter – it will be AccessType.PROPERTY.
* If you set @Access on the class/entity level, JPA accesses all properties of the class according to the selected strategy irrespective of place of Id/EmeddedId annotation.
* When using Field Access Mode, all fields must be declared as either protected, package, or private. Public fields are disallowed because it would open up the state fields to access by any unprotected class in the JVM.
* When using Property Access Mode, both methods must be either public or protected visibility. The mapping annotations for a property must be on the getter method.

Example:

@Entity

@Access(AccessType.FIELD)

public class Student { … }

**Spring Data Validations**

1. **@NotNull:** Field must not be null (but can be empty for collections/strings).

Example:

@NotNull(message = "Name cannot be null")

private String name;

1. **@NotEmpty:** Field must not be null or empty (applies to strings, collections).

Example:

@NotEmpty(message = "List must not be empty")

private List<String> tags;

1. **@NotBlank:** Field must not be null, empty, or only whitespace (for Strings).

Example:

@NotBlank(message = "Username cannot be blank")

private String username;

1. **@Size:** Validates that a string/collection/array is within a specific size range.

Example:

@Size(min = 2, max = 30, message = "Name must be between 2 and 30 characters")

private String name;

1. **@Min and @Max:** Validates that a number is greater than or equal to (@Min) or less than or equal to (@Max) the specified value.

Example:

@Min(value = 18, message = "Age should be at least 18")

@Max(value = 60, message = "Age should not be more than 60")

private int age;

1. **@Email**: Validates that the field is a well-formed email address.

Example:

@Email(message = "Invalid email address")

private String email;

1. **@Pattern:** Validates that a string matches the given regex.

Example:

@Pattern(regexp = "\\d{10}", message = "Phone number must be 10 digits")

private String phoneNumber;

1. **@Positive / @PositiveOrZero:** Ensures a number is positive (>0) or positive including zero (>=0).

Example:

@Positive(message = "Price must be greater than 0")

private double price;

1. **@Negative / @NegativeOrZero:** Ensures a number is negative (<0) or negative including zero (<=0).

Example:

@Negative(message = "Value must be negative")

private int value;

1. **@AssertTrue / @AssertFalse:** Field must be true or false.

Example:

@AssertTrue(message = "You must accept the terms")

private boolean acceptedTerms;

1. **@Future / @FutureOrPresent:** Validates that a date is in the future or in the future or present.

Example:

@Future(message = "Date must be in the future")

private LocalDate bookingDate;

1. **@Past / @PastOrPresent:** Validates that a date is in the past or in the past or present.

Example:

@PastOrPresent(message = "Date of birth cannot be in the future")

private LocalDate dob;

1. **@Digits(integer, fraction):** Validates that a number has a specific number of integer and fractional digits.

Example:

@Digits(integer = 6, fraction = 2, message = "Salary must be a valid amount")

private BigDecimal salary;

1. **@Valid:** Used to cascade validation to nested objects.

Example:

@Valid

private Address address;

1. **@Null:** Field must be null (useful for controlled APIs where field must not be provided).

Example:

@Null(message = "ID must not be provided when creating a new user")

private Long id;

**API Response**

1. **Using a DTO as return object:**

@GetMapping("/example")

public MyResponseObject getExample() {

return new MyResponseObject("Success", 200);

}

1. **Returning String:**

@GetMapping("/hello")

public String sayHello() {

return "Hello, World!";

}

1. **Returning void:**

@PostMapping("/example")

public void processExample(@RequestBody MyRequestObject request) {

// processing logic, no content to return

}

1. **Returning a Map:**

@GetMapping("/map-response")

public Map<String, Object> getMapResponse() {

Map<String, Object> response = new HashMap<>();

response.put("message", "Success");

response.put("status", 200);

return response;

}

1. **Returning a list of DTOs:**

@GetMapping("/list-response")

public List<MyResponseObject> getListResponse() {

return Arrays.asList(

new MyResponseObject("Item 1", 1),

new MyResponseObject("Item 2", 2)

);

}

1. **Using ResponseEntity: It gives us more control over the HTTP response, allowing us to set the status code, headers, and body explicitly.**

@GetMapping("/example")

public ResponseEntity<MyResponseObject> getExample() {

MyResponseObject response = new MyResponseObject("Success", 200);

return new ResponseEntity<>(response, HttpStatus.OK);

}

@GetMapping("/example")

public ResponseEntity<MyResponseObject> getExample() {

MyResponseObject response = new MyResponseObject("Success", 200);

HttpHeaders headers = new HttpHeaders();

headers.add("Custom-Header", "Value");

return new ResponseEntity<>(response, headers, HttpStatus.OK);

}

@PostMapping("/example")

public ResponseEntity<Void> processExample(@RequestBody MyRequestObject request) {

// processing logic, no content to return

return new ResponseEntity<>(HttpStatus.NO\_CONTENT); // 204 No Content

}

@GetMapping("/file")

public ResponseEntity<byte[]> downloadFile() throws IOException {

File file = new File("example.pdf");

byte[] contents = Files.readAllBytes(file.toPath());

HttpHeaders headers = new HttpHeaders();

headers.setContentType(MediaType.APPLICATION\_PDF);

headers.setContentDispositionFormData("attachment", "example.pdf");

return new ResponseEntity<>(contents, headers, HttpStatus.OK);

}

**Global Exception Handling**

**Implementing a global exception handler in a Spring Boot application is done using @ControllerAdvice along with @ExceptionHandler.**

**@ControllerAdvice** is a specialization of @Component + AOP-like behavior. It works across all @Controller and @RestController classes.

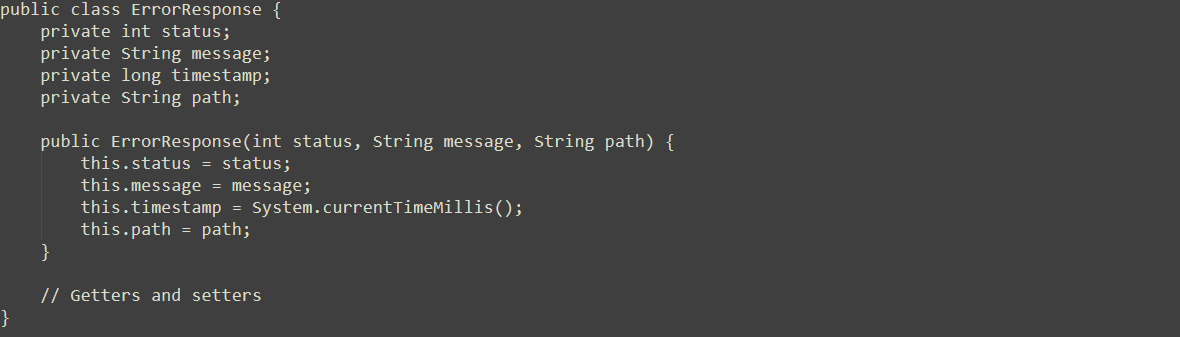
**@RestControllerAdvice:** A convenience annotation that combines @ControllerAdvice and @ResponseBody. Same as above but returns JSON/XML by default.

**Exception Propagation Flow in Spring Boot:** Lifecycle Overview

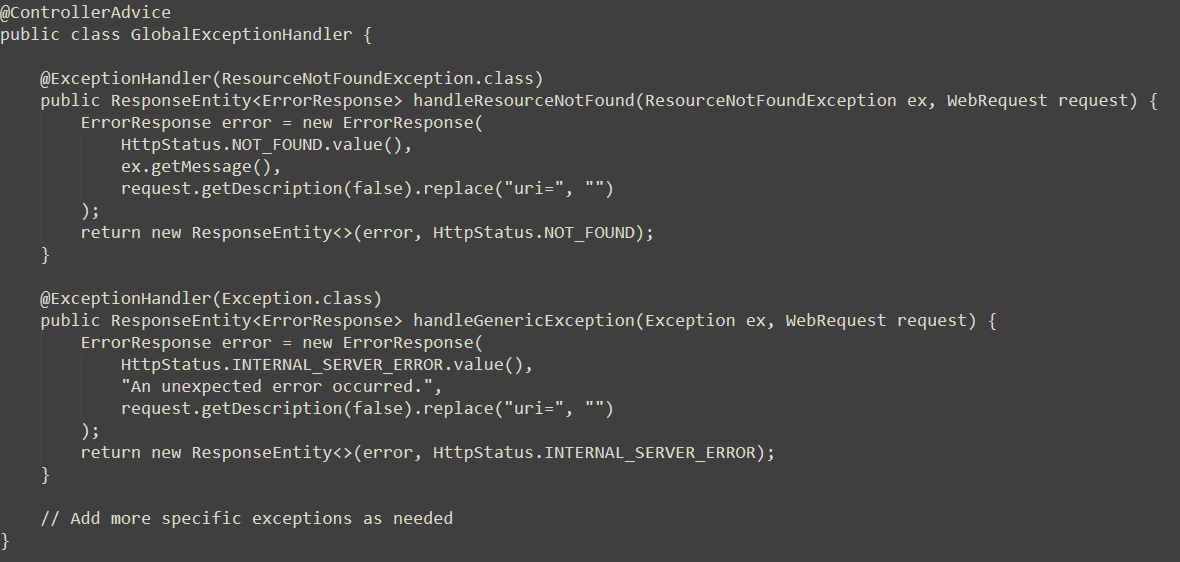
1. Request comes in → hits a @Controller or @RestController. The request is received by Spring's DispatcherServlet, which delegates it to the appropriate controller method.
2. If something goes wrong (exception is thrown) during any of the following: Controller execution, Service layer logic, Validation (e.g., @Valid, @Validated), Data access layer (e.g., repository/database errors) an uncaught exception propagates up the call stack.
3. Spring’s DispatcherServlet catches the unhandled exception. The DispatcherServlet acts as the central point of exception resolution.
4. Spring checks for a matching @ExceptionHandler, first, it looks in the same controller class where the exception occurred. Next, it checks any global exception handler defined with @ControllerAdvice or @RestControllerAdvice.
5. If a matching @ExceptionHandler is found, That method is invoked. The handler builds an appropriate response (e.g., using ResponseEntity, custom error body). If no matching handler is found, Spring falls back to its default exception resolution mechanism.

Example:

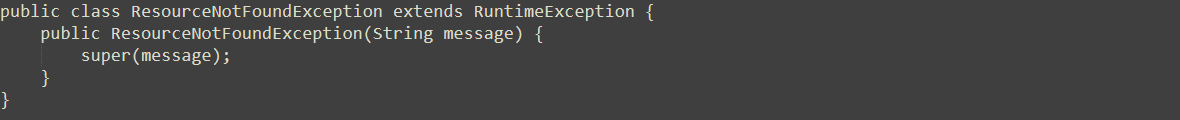
Step1: Create a Custom Error Response Class.



Step2: Define the Global Exception Handler using @ControllerAdvice



Step3: (Optional) Define Custom Exceptions



**Caching**

**Enabling Caching in a Spring Boot Application:**

Example:

@SpringBootApplication

@EnableCaching

public class MyApplication {

public static void main(String[] args) {

SpringApplication.run(MyApplication.class, args);

}

}

**Method-Level Caching**

With method level caching the results of method calls are stored so that subsequent calls with the same parameters can be served from the cache rather than executing the method again.

Spring Boot provides built-in support for method-level caching via the Spring Cache Abstraction using an in-memory cache (based on ConcurrentHashMap) unless you explicitly configure another cache provider (like Redis).

Core Caching Annotations:

**1.@Cacheable:** The method runs only the first time; later calls return from cache. Use for read operations.

Example:

@Cacheable("users")

public User getUserById(Long id) {

System.out.println("Fetching user from DB...");

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

}

"users" is the cache name. The cache key defaults to the method parameter (i.e., id).

**2.@CachePut:** This forces the method to execute every time. It updates the cache with the latest return value. Use for write/update operations to refresh the cache.

Example:

@CachePut(value = "users", key = "#user.id")

public User updateUser(User user) {

return userRepository.save(user);

}

**3.@CacheEvict:** Removes the cached value for the given key when the method is called. Use when you delete or invalidate data.

Example:

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

public void deleteUser(Long id) {

userRepository.deleteById(id);

}

@CacheEvict(value = "users", allEntries = true)

public void clearCache() {

// Clears all cached entries

}

**Web Layer (HTTP) Caching**

Caching the HTTP responses at the controller or web server level to avoid regenerating the same response for repeated requests.

Two Main Ways to Implement Web Layer Caching:

**1.HTTP Response Caching Using Headers (Client/Proxy-Side):**

This method uses HTTP headers to instruct browsers or proxies (like Cloudflare, NGINX) to cache responses.

Example:

Cache-Control: max-age=60

Instructs browsers and proxies to cache the response for 60 seconds.

**2.Server-Side Caching in Controllers:** Spring Cache (with @Cacheable)

**Scheduling & Async**

1. **@Scheduled**

* Used to execute a method on a scheduled basis.
* Before using @Scheduled, you must enable scheduling in your Spring Boot application using @EnableScheduling.
* By default, all scheduled tasks share a single-threaded executor. If one task blocks, others wait.

Example:

@SpringBootApplication

@EnableScheduling

public class MyApp {

public static void main(String[] args) {

SpringApplication.run(MyApp.class, args);

}

}

@Component

public class MyScheduler {

@Scheduled(fixedRate = 5000)

public void runTask() {

System.out.println("Running every 5 seconds: " + LocalDateTime.now());

}

}

You can use 4 different scheduling styles:

1. fixedRate - runs at a regular interval

2. fixedDelay - runs after a delay between method completions

3. initialDelay with fixedRate or fixedDelay - Starts the initial execution after app startup, then continues every fixed seconds.

4. cron - provides a cron format ‘second minute hour day-of-month month day-of-week year(optional)’

**API Level Security**

1. **@EnableWebSecurity**

* Enables Spring Security’s web security support and allows customization of security configurations via **SecurityFilterChain**.

Example:

@Configuration

@EnableWebSecurity

public class SecurityConfig {

@Bean

public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

return http

.csrf(csrf -> csrf.disable()) // Disable for stateless APIs

.sessionManagement(session ->

session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))

.authorizeHttpRequests(auth -> auth

.requestMatchers("/api/public/\*\*").permitAll()

.requestMatchers("/api/admin/\*\*").hasRole("ADMIN")

.requestMatchers("/api/user/\*\*").hasAnyRole("USER", "ADMIN")

.anyRequest().authenticated()

)

.addFilterBefore(new JwtAuthenticationFilter(jwtService),

UsernamePasswordAuthenticationFilter.class)

.build();

}

}

1. **@EnableMethodSecurity**

**To secure individual methods in your Spring Boot REST controllers or service classes, you must enable method-level security using the @EnableMethodSecurity annotation.**

* Class level annotation. Used in a configuration class, usually the same one where you define SecurityFilterChain.
* Tells Spring to intercept method calls and apply security checks.
* Works in combination with using annotations like @PreAuthorize, @PostAuthorize, @Secured, and @RolesAllowed.
* prePostEnabled = true: Enables @PreAuthorize and @PostAuthorize
* securedEnabled = true: Enables @Secured
* jsr250Enabled = true: Enables @RolesAllowed

Example:

@Configuration

@EnableWebSecurity

@EnableMethodSecurity(prePostEnabled = true)

public class SecurityConfig {

// define beans like SecurityFilterChain

}

1. **@PreAuthorize**

* Allows to define access control rules before a method is executed.
* It uses Spring Expression Language (SpEL) to define complex authorization logic such as checking roles, authorities, custom conditions, and even method parameters.
* It can be applied on controller methods or service layer methods.

Example:

@PreAuthorize("hasRole('ADMIN') and hasAuthority('USER\_READ')")

public void deleteUser(Long id) {}

1. **@PostAuthorize**

* Performs authorization checks after a method has executed but before the result is returned to the caller.
* Useful when you want to allow a method to execute but restrict who can see the result.
* An AccessDeniedException is thrown and the result is blocked.

Example:

@PostAuthorize("hasRole('ADMIN') or returnObject.owner == authentication.name")

public Document fetchDocument(Long id) {

return documentService.find(id);

}

1. **@Secured**

* Used to restrict access to a method based on the user's role(s).
* Role names must be prefixed with ROLE\_

Example:

@Secured({"ROLE\_ADMIN", "ROLE\_MANAGER"})

public void viewReports() {}

1. **@RolesAllowed**

* It's similar to @Secured, but not tied to Spring — it's a Java standard that Spring Security supports.

Example:

@RolesAllowed({"ADMIN", "MANAGER"})

public void accessDashboard() {}