SPRINGBOOT FRAMEWORK

Represented By Ajay Kumar

Ajay Kumar – LinkedIn Profile: https://www.linkedin.com/in/ajay-kumar-sahni-4a02322b8/

Q:
What is Spring Boot ?

- **"Spring Boot is a framework built on top of Spring that reduces configuration efforts using auto-configuration, starter dependencies, and embedded servers ".**

- **It provides production-ready features such as monitoring, health checks, and metrics.**

Q:
"Why do we need Spring Boot?"

- **"Because configuring Spring projects manually with XML/Java configs was complex and time-consuming. Spring Boot removes that burden with auto-configuration and embedded servers."**

Q:
What's Differences between Spring Boot and Spring Framework ?

Feature	Spring	Spring Boot
Setup	Requires complex **XML/Java configuration**	**Auto-configuration,** minimal setup
Deployment	Needs **external server (Tomcat/Jetty)**	**Embedded servers (Tomcat, Jetty, Undertow)**
Dependencies	Need to manage versions manually	Provides **starter dependencies**
Production-ready	Limited support	Actuator, health checks, metrics
Learning curve	Steeper	Easier

Q:

Explain about Spring Boot Architecture ?

- "Spring Boot architecture is layered: Controller → Service → Repository → Database.

- Internally, it uses @SpringBootApplication for auto-configuration, an embedded server (Tomcat/Jetty), and DispatcherServlet for request handling, making application development and deployment faster."

I . High-Level Architecture :

==> Spring Boot consists of the following main parts:

1. **Spring Core Framework** (Dependency Injection, Bean management).

2. **Auto-Configuration** (configures beans automatically based on classpath).

3. **Starter Dependencies** (predefined dependencies for easy setup).

4. **Embedded Servers** (Tomcat, Jetty, Undertow – no need for external deployment).

5. **Spring Boot CLI** (for rapid prototyping with Groovy).

6. **Spring Boot Actuator** (production-ready features like monitoring, metrics, and health checks).

II . Layered Architecture (Flow of Application) :

==> Spring Boot follows a multi-layered architecture:

1. Presentation Layer (Controller Layer)

- Handles HTTP requests (using **@Controller** or **@RestController**).

- Sends responses back (JSON, XML, HTML).

- Works with Spring MVC (DispatcherServlet).

#. Example:

```
@RestController
@RequestMapping("/api")
 public class UserController {
@GetMapping("/users")
 public List<User> getUsers() {
      return List.of(new User(1, "Ajay"), new User(2, "Kumar"));
  }
}
```

2. Business Layer (Service Layer)

- Contains business logic.
- Uses **@Service** annotation.
- Calls the repository layer for data.

3. Persistence Layer (Repository Layer)

- Interacts with the database.
- Uses Spring Data JPA / Hibernate.
- Annotated with @Repository.

#. Example:

```
@Repository
    public interface UserRepository extends JpaRepository<User, Integer> {

    }
```

4. Database Layer

- Actual database (MySQL, PostgreSQL, Oracle, MongoDB, etc.).
- Mapped using JPA + Hibernate ORM.

✅ Flow in Spring Boot:

==> **Client → Controller → Service → Repository → Database → Response**

III . Internal Architecture of Spring Boot

==> Now, from a deeper interview angle, here's how things work internally:

1. @SpringBootApplication

- **Combines:**
  - **@Configuration** (marks class as configuration source).
  - **@EnableAutoConfiguration** (enables Spring Boot's auto-configuration mechanism.).
  - **@ComponentScan** (scans the package and sub-packages for beans (@Component, @Service, @Repository, @Controller)) .

Q: How does @SpringBootApplication work internally?

Internal Process:

- When you call SpringApplication.run(App.class, args):
  - It creates an ApplicationContext (Spring container).
  - Loads beans defined by configuration and component scan.
  - Triggers auto-configuration by reading META-INF/spring.factories.
  - Starts the embedded server (Tomcat/Jetty).
  - Deploys DispatcherServlet for handling requests.

✅ **Interview answer (short):**

- "@SpringBootApplication is a convenience annotation that combines @Configuration, @EnableAutoConfiguration, and @ComponentScan. It tells Spring Boot to scan for beans, apply auto-configuration, and bootstrap the application with an embedded server."

2. Auto-Configuration

Q: What is auto-configuration and how does it decide which beans to load?

- Auto-Configuration is one of Spring Boot's core features.
- **It configures beans automatically based on:**
  Classpath contents (e.g., if Hibernate JAR exists → configure JPA beans).
  Existing beans (if no custom DataSource bean, create a default one).
  Property files (application.properties/yaml).

Internals:

- Controlled by @EnableAutoConfiguration.
- Uses spring.factories (inside JARs like spring-boot-autoconfigure).
- **Example:**
  - If spring-boot-starter-web is present → load DispatcherServlet, Jackson, and Tomcat.
  - If spring-boot-starter-data-jpa is present → load EntityManagerFactory, Hibernate, DataSource.
  - **Example:**
    - If spring-boot-starter-web is present → auto-configures DispatcherServlet, Jackson, and Tomcat.
    - If spring-boot-starter-data-jpa is present → configures EntityManagerFactory, Hibernate, TransactionManager.

✅ **Interview answer (short):**

- "Auto-configuration is enabled by **@EnableAutoConfiguration** and works by scanning **spring.factories.** It uses conditions like **@ConditionalOnClass** and **@ConditionalOnMissingBean** to load beans only if required dependencies are present and no custom bean is defined."

3. Embedded Servers

- Tomcat/Jetty/Undertow run inside the JAR file.
- No need for WAR + external deployment.
- Entry point is SpringApplication.run().

4. DispatcherServlet (Spring MVC)

Q: How is DispatcherServlet involved in request handling?

- DispatcherServlet is the Front Controller in Spring MVC.
- **In Spring Boot:**
  - It is auto-configured when spring-boot-starter-web is present.
  - Mapped to handle all requests (/).

==> Flow of request handling:

1. Client sends HTTP request → handled by DispatcherServlet.
2. DispatcherServlet consults HandlerMapping → finds the appropriate Controller.
3. Controller executes business logic (via Service layer).
4. Returns ModelAndView/ResponseEntity → processed by ViewResolver/HttpMessageConverter.
5. DispatcherServlet sends final response to client.

✅ **Interview answer (short):**

- "DispatcherServlet acts as the front controller in Spring Boot. It intercepts all incoming HTTP requests, maps them to the appropriate controller method using HandlerMappings, executes business logic, and sends back the response using ViewResolvers or HttpMessageConverters."

Q:
How does Spring Boot differ from traditional Spring MVC in terms of deployment?

Feature	Traditional Spring MVC	Spring Boot
Server	Needs external server (Tomcat/Jetty). Deploy WAR file manually.	Has **embedded server** (Tomcat/Jetty/Undertow). Runs as a JAR (java -jar).
Configuration	XML + Java Config needed	Minimal config → auto-configuration + starters
Dependency Management	Developer manages versions	**Spring Boot starters** manage compatible versions
Deployment	Manual WAR deployment in webapps/	Direct execution → mvn spring-boot:run or java -jar
Monitoring	Needs custom setup	Built-in **Actuator** endpoints

✅ Interview answer (short):

- "In traditional Spring MVC, you need to package the app as a WAR and deploy it on an external server. In Spring Boot, applications run as standalone JARs with embedded servers like Tomcat, so no external deployment is required."

#. Starter Dependencies (spring-boot-starter-*)

- Starter dependencies are a set of pre-configured Maven/Gradle dependencies provided by Spring Boot.
- They bundle together commonly used libraries and their compatible versions.
- Avoids version hell → Spring Boot chooses compatible versions automatically.
- Purpose → To simplify dependency management and avoid version conflicts.

Q: Common Spring Boot Starter Dependencies

Starter Dependency	Purpose	Includes (Key Libraries)	Real-World Use Case
spring-boot-starter	Core starter	Logging (Logback), YAML, Auto-config support	Base dependency for any Spring Boot app
spring-boot-starter-web	Build REST APIs & MVC apps	Spring MVC, Jackson, Embedded Tomcat	RESTful APIs, Microservices, Web apps
spring-boot-starter-data-jpa	Relational DB with JPA	Hibernate, JPA, HikariCP	CRUD apps, E-commerce DB, Banking apps
spring-boot-starter-data-mongodb	NoSQL DB	Spring Data MongoDB	Chat apps, Real-time analytics
spring-boot-starter-security	Authentication & Authorization	Spring Security	Login system, Role-based access control
spring-boot-starter-thymeleaf	Server-side templates	Thymeleaf template engine	Dynamic HTML views (Admin dashboards)
spring-boot-starter-test	Testing framework	JUnit, Mockito, Spring Test, Hamcrest	Unit & integration testing
spring-boot-starter-validation	Bean validation	Hibernate Validator (JSR-380)	Validating user input (forms, API requests)
spring-boot-starter-mail	Email support	JavaMail API	Sending OTPs, notifications
spring-boot-starter-actuator	Monitoring & management	Actuator endpoints	Health checks, Metrics, Production monitoring
spring-boot-starter-cache	Caching support	JCache, EhCache, Redis support	Performance boost in e-commerce, caching products
spring-boot-starter-aop	Aspect-Oriented Programming	Spring AOP, AspectJ	Logging, Transaction management
spring-boot-starter-batch	Batch processing	Spring Batch	Payroll, report generation
spring-boot-starter-quartz	Scheduling tasks	Quartz Scheduler	Cron jobs, notifications, cleanup tasks
spring-boot-starter-webflux	Reactive Web Apps	Spring WebFlux, Reactor	Real-time apps, Streaming APIs
spring-boot-starter-integration	Enterprise integration	Spring Integration	Messaging, API orchestration

#. CRUD operations with JPA Repository in Spring Boot

1. Entity Class :

```
package com.example.demo.entity;
import jakarta.persistence.*;

@Entity
@Table(name = "employees")      // Table name in DB
public class Employee {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)      // Auto-increment ID
    private Long id;

    @Column(nullable = false)
    private String name;

    private String department;

    private Double salary;

  // Constructors
    public Employee() {
     }
    public Employee(String name, String department, Double salary) {
        this.name = name;
        this.department = department;
        this.salary = salary;
    }

  // Getters & Setters
    public Long getId() {
        return id;
     }
    public void setId(Long id) {
        this.id = id;
     }

    public String getName() {
         return name;
     }
```

```
 public void setName(String name) {
      this.name = name;
  }
 public String getDepartment() {
    return department;
  }
 public void setDepartment(String department) {
     this.department = department;
  }

 public Double getSalary() {
    return salary;
  }
 public void setSalary(Double salary) {
     this.salary = salary;
   }
}
```

2. application.properties

```
spring.datasource.url=jdbc:mysql://localhost:3306/testdb
spring.datasource.username=root
spring.datasource.password=yourpassword
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

3. Repository Interface

```
package com.example.demo.repository;

import com.example.demo.entity.Employee;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface EmployeeRepository extends JpaRepository<Employee, Long> {

        // You can also define custom queries here

       // Example: List<Employee> findByDepartment(String department);
}
```

4. Service Layer

```
package com.example.demo.service;
import com.example.demo.entity.Employee;
import com.example.demo.repository.EmployeeRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import java.util.List;

@Service
public class EmployeeService {

    @Autowired
    private EmployeeRepository employeeRepository;

    // CREATE
    public Employee saveEmployee(Employee employee) {
        return employeeRepository.save(employee);
    }

    // READ (all employees)
    public List<Employee> getAllEmployees() {
        return employeeRepository.findAll();
    }

    // READ (by ID)
    public Employee getEmployeeById(Long id) {
        return employeeRepository.findById(id).orElse(null);
    }

    // UPDATE
    public Employee updateEmployee(Long id, Employee employeeDetails) {
        Employee existing = employeeRepository.findById(id).orElse(null);
        if (existing != null) {
            existing.setName(employeeDetails.getName());
            existing.setDepartment(employeeDetails.getDepartment());
            existing.setSalary(employeeDetails.getSalary());
            return employeeRepository.save(existing);
        }
        return null;
    }
    // DELETE
    public String deleteEmployee(Long id) {
        employeeRepository.deleteById(id);
        return "Employee deleted successfully!";
    }
}
```

5. REST Controller

```
package com.example.demo.controller;
import com.example.demo.entity.Employee;
import com.example.demo.service.EmployeeService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import java.util.List;

@RestController
@RequestMapping("/employees")
public class EmployeeController {

    @Autowired
    private EmployeeService employeeService;

    // CREATE
    @PostMapping
    public Employee createEmployee(@RequestBody Employee employee) {
        return employeeService.saveEmployee(employee);
    }

    // READ all
    @GetMapping
    public List<Employee> getAllEmployees() {
        return employeeService.getAllEmployees();
    }

    // READ by ID
    @GetMapping("/{id}")
    public Employee getEmployeeById(@PathVariable Long id) {
        return employeeService.getEmployeeById(id);
    }

    // UPDATE
    @PutMapping("/{id}")
    public Employee updateEmployee(@PathVariable Long id, @RequestBody Employee
employee) {
        return employeeService.updateEmployee(id, employee);
    }

    // DELETE
    @DeleteMapping("/{id}")
    public String deleteEmployee(@PathVariable Long id) {
        return employeeService.deleteEmployee(id);
    }
}
```

Testing CRUD with Postman

1. POST → http://localhost:8080/employees

Body (JSON):

```
{
    "name": "Ajay Kumar",
    "department": "IT",
    "salary": 55000
 }
```

2. GET → http://localhost:8080/employees (all employees)

3. GET → http://localhost:8080/employees/1 (by ID)

4. PUT → http://localhost:8080/employees/1

Body:

```
{
   "name": "Ajay Kumar Sahni",
    "department": "Full Stack",
    "salary": 60000
 }
```

5. DELETE → http://localhost:8080/employees/1

#. Annotations in Springboot :

1. **@SpringBootApplication** :
   - "@SpringBootApplication is a convenience annotation that combines @Configuration, @EnableAutoConfiguration, and @ComponentScan. It tells Spring Boot to auto-configure the application based on the classpath dependencies, scan for components, and register them as beans. It acts as the entry point of a Spring Boot application."

Q:
What's Differences betweeen @Controller and @RestController in Springboot ?

@Controller :

- @Controller is used to define web controllers that return views .

@RestController :

- @RestController is used for RESTful web services that return data directly as JSON or XML.

- Technically, @RestController is a combination of @Controller and @ResponseBody, which means all its handler methods return data instead of views."

Feature	@Controller	@RestController
Purpose	Used to define a controller that returns **views (HTML, JSP, Thymeleaf)**.	Used to define a RESTful controller that returns **data (JSON/XML)** directly.
Annotation Type	It is a **specialized version of @Component** that marks a class as a Spring MVC controller.	It is a **combination of @Controller + @ResponseBody**.
Return Type Handling	Returns **view name**, which is resolved by **ViewResolver** (e.g., Thymeleaf, JSP).	Returns **data (objects)** which are automatically **serialized to JSON or XML**.
Use Case	Used in **Web MVC applications** that serve dynamic web pages.	Used in **RESTful APIs** that serve data to clients like Angular, React, or mobile apps.
Need for @ResponseBody	You must add **@ResponseBody** manually to send JSON data.	@RestController already includes @ResponseBody, so no need to add it.

⚙ Internal Difference

🔸 @Controller :

Internally, it behaves like this:

Example :

```
@Controller
public class MyController {

   @RequestMapping("/welcome")
   public String welcome() {
      return "welcome";          // returns view name (welcome.html or welcome.jsp)
   }
   @RequestMapping("/data")
   @ResponseBody
   public String getData() {
      return "Hello User";       // returns plain text or JSON (because of @ResponseBody)
   }
}
```

Note :

- If you don't use @ResponseBody, it assumes you are returning a view name.
- Uses ViewResolver to map the return value to an HTML/JSP page.

🔹 @RestController :

Internally, it's defined as:

Example :

```
@RestController
public class MyRestController {

   @GetMapping("/hello")
   public String sayHello() {
      return "Hello Ajay";          // directly returns data as response body (JSON/text)
   }
   @GetMapping("/user")
   public User getUser() {
      return new User(1, "Ajay", "Backend Developer");
   }
}
```

#. Response Output (JSON):

```
{
  "id": 1,
  "name": "Ajay",
  "role": "Backend Developer"
}
```

- No **@ResponseBody** needed — Spring Boot automatically converts objects into JSON using Jackson.

4. @RequestMapping :

- @RequestMapping is used to map web requests (HTTP requests) to specific controller methods or classes in a Spring MVC or Spring Boot application.

- It acts as a bridge between the client (HTTP request) and the backend controller method that handles it.

🔹 Basic Syntax

```
@RequestMapping(value = "/path", method = RequestMethod.GET)
 public String handlerMethod() {
      return "response";
}
```

Here:

- /path → URL endpoint.
- RequestMethod.GET → type of HTTP request.
- The method executes when the client sends a GET request to /path.

Used On:

1. Class level: defines a base URL for all methods.

2. Method level: defines the specific endpoint.

Example :

```
@RestController
@RequestMapping("/api/users")    // Base URL for all endpoints in this controller
public class UserController {

   @RequestMapping(value = "/getAll", method = RequestMethod.GET)
   public List<User> getAllUsers() {
      return List.of(new User(1, "Ajay"), new User(2, "Rahul"));
   }

   @RequestMapping(value = "/save", method = RequestMethod.POST)
   public String saveUser(@RequestBody User user) {
      return "User saved: " + user.getName();
   }
}
```

Endpoints:

- **GET /api/users/getAll**

- **POST /api/users/save**

#. Shortcut Annotations (Spring 4.3+)

- **Spring introduced specialized versions of @RequestMapping for readability and clarity:**

Shortcut	Equivalent to	Used for
@GetMapping	@RequestMapping(method = RequestMethod.GET)	GET requests
@PostMapping	@RequestMapping(method = RequestMethod.POST)	POST requests
@PutMapping	@RequestMapping(method = RequestMethod.PUT)	PUT requests
@DeleteMapping	@RequestMapping(method = RequestMethod.DELETE)	DELETE requests
@PatchMapping	@RequestMapping(method = RequestMethod.PATCH)	PATCH requests

Q: Internally — How @RequestMapping Works

1. When Spring Boot starts, the DispatcherServlet scans all controllers.

2. It looks for methods annotated with @RequestMapping (or @GetMapping, etc.).

3. It registers each endpoint in a HandlerMapping (a mapping table between URLs and controller methods).

4. When a request arrives:
   - The DispatcherServlet matches the URL + HTTP method.
   - Calls the appropriate method.
   - The return value is processed by ViewResolver or MessageConverter (for JSON/XML).

Q: "What is @RequestMapping and how does it work?"

- "@RequestMapping is used to map HTTP requests to handler methods in Spring Boot. It can be used at both class and method levels. It supports specifying URL paths, HTTP methods, headers, and content types. The DispatcherServlet uses this mapping information to route incoming requests to the correct controller method."

#. Supported Attributes

- **You can customize your request mapping using attributes:**

Attribute	Description	Example
value or path	URL path	@RequestMapping("/users")
method	HTTP method	@RequestMapping(method = RequestMethod.GET)
produces	MIME type response	@RequestMapping(produces = "application/json")
consumes	Accepted request type	@RequestMapping(consumes = "application/json")
params	Match based on query parameters	@RequestMapping(params = "version=1")
headers	Match based on headers	@RequestMapping(headers = "content-type=application/json")

5. @GetMapping?

- @GetMapping is a **composed annotation** in Spring Boot that acts as a shortcut for:
  @RequestMapping(method = RequestMethod.GET)
- It is used to handle HTTP GET requests — i.e., when the client wants to retrieve data from the server.

Example :

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

   @GetMapping("/all")
   public List<User> getAllUsers() {
      return List.of(
         new User(1, "Ajay", "Backend Developer"),
         new User(2, "Rahul", "Frontend Developer") );
   }
}
```

#. When a client hits: **GET http://localhost:8080/api/users/all**

then Output (JSON):

```
[
  {"id": 1, "name": "Ajay", "role": "Backend Developer"},
  {"id": 2, "name": "Rahul", "role": "Frontend Developer"}
]
```

Q:
Why use @GetMapping instead of @RequestMapping?

- **@GetMapping** is more specific, readable, and modern.

#. Instead of writing:

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

#. You simply write:

```
@GetMapping("/all")
```

- This makes code cleaner and easier to maintain — especially when your API has multiple HTTP methods (GET, POST, PUT, DELETE, etc.).

⚙ Internal Working

When the Spring Boot application starts:

1. The DispatcherServlet scans all controllers.
2. It finds methods annotated with @GetMapping.
3. It registers them in a HandlerMapping table.
4. When a GET request comes, it:
   - Matches the URL pattern.
   - Calls the mapped method.
   - Uses MessageConverter (like Jackson) to convert the Java object → JSON response.

#. Common Attributes of @GetMapping

Attribute	Description	Example
value / path	URL path	@GetMapping("/users")
produces	Type of response (MIME type)	@GetMapping(value="/users", produces="application/json")
params	Filter by request parameters	@GetMapping(value="/users", params="version=1")
headers	Match specific headers	@GetMapping(headers="content-type=application/json")

1. Example with Path Variables :

```
@GetMapping("/user/{id}")
 public User getUserById(@PathVariable int id) {
     return new User(id, "Ajay", "Developer");
 }
```

Request:

GET /api/user/10

Response:

{"id":10,"name":"Ajay","role":"Developer"}

2. Example with Query Parameters

```
@GetMapping("/search")
 public String searchUser(@RequestParam String name) {
     return "Searching for user: " + name;
  }
```

Request:

GET /api/search?name=Ajay

Response:

Searching for user: Ajay

Q:

"What is @GetMapping and how is it different from @RequestMapping?"

- "@GetMapping is a composed annotation introduced in Spring 4.3 that acts as a shortcut for @RequestMapping(method = RequestMethod.GET).

- It is used to handle HTTP GET requests in RESTful services. It improves readability and clarity compared to @RequestMapping, which requires specifying the method explicitly."

Then you can add:

- "Internally, it's handled by the DispatcherServlet, which maps GET requests to the appropriate handler method and returns the response in JSON or XML format."

Q: What's Differences between @GetMapping and @RequestMapping ?

Aspect	@GetMapping	@RequestMapping
Purpose	Handle HTTP GET requests	Handle any HTTP method
Introduced in	Spring 4.3	Core Spring MVC
Syntax	@GetMapping("/path")	@RequestMapping(value="/path", method=RequestMethod.GET)
Readability	More readable & modern	Verbose
Use Case	Retrieving data	Any kind of request

6. @PostMapping :

- **@PostMapping** is a composed annotation in Spring Boot used to handle HTTP POST requests.

It acts as a shortcut for :

```
@RequestMapping(method = RequestMethod.POST)
```

- It's generally used when the client sends data to the server — usually to create a new resource (for example: adding a user, saving employee data, registering an account, etc.)

Example :

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

@PostMapping("/add")
 public String addUser(@RequestBody User user) {
        return "User added: " + user.getName();
  }
}
```

Request:

POST → http://localhost:8080/api/users/add

Request Body (JSON):

```
{
  "id": 1,
  "name": "Ajay",
  "role": "Backend Developer"
}
```

Response:

User added: Ajay

Q:

How It Works Internally ?

1. When Spring Boot starts, the DispatcherServlet scans all controllers.

2. It detects @PostMapping annotations and registers them in the HandlerMapping.

3. **When a POST request is received:**
   - The URL and HTTP method are matched.
   - The corresponding controller method executes.
   - The @RequestBody converts JSON → Java object using the Jackson library.
   - The return value is converted back to JSON and sent to the client.

Example with ResponseEntity :

- To return proper HTTP status and messages:

Example :

```
@PostMapping("/save")
public ResponseEntity<User> saveUser(@RequestBody User user) {

    User savedUser = userService.save(user);

        return ResponseEntity.status(HttpStatus.CREATED).body(savedUser);
}
```

Response (Status 201 Created):

```
{
 "id": 1,
 "name": "Ajay",
 "role": "Backend Developer"
}
```

Q:
What's Difference Between @GetMapping and @PostMapping ?

Feature	@GetMapping	@PostMapping
HTTP Method	GET	POST
Purpose	Retrieve data	Create or submit data
Request Body	Not allowed (parameters in URL)	Allowed (JSON/XML in body)
Caching	Usually cacheable	Not cacheable
Idempotent	Yes (safe to call multiple times)	No (creates new resource each time)

Full Example: POST + GET Together :

```
@RestController
@RequestMapping("/api/employees")
public class EmployeeController {

    private List<Employee> employees = new ArrayList<>();

    @PostMapping("/add")
    public Employee addEmployee(@RequestBody Employee emp) {

        employees.add(emp);

        return emp;
    }

    @GetMapping("/all")
    public List<Employee> getAllEmployees() {

         return employees;
    }
}
```

Test:

1. **POST → /api/employees/add**
   {"id": 101, "name": "Ajay", "role": "Backend Developer"}

2. **GET → /api/employees/all**
   [{"id": 101, "name": "Ajay", "role": "Backend Developer"}]

7. @PutMapping :

- **@PutMapping** is a composed annotation in Spring Boot that is used to handle HTTP PUT requests.

It's a shortcut for:

@RequestMapping(method = RequestMethod.PUT)

#. Purpose:

- It is Used to update an existing resource in database via api (for example, updating user details, changing employee info, etc.)

Basic Example :

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

    private Map<Integer, User> users = new HashMap<>();

    @PutMapping("/update/{id}")
    public ResponseEntity<User> updateUser(@PathVariable int id, @RequestBody User updatedUser) {

        User existingUser = users.get(id);

        if (existingUser == null) {
            return ResponseEntity.notFound().build();
        }

        existingUser.setName(updatedUser.getName());

        existingUser.setRole(updatedUser.getRole());

        users.put(id, existingUser);

        return ResponseEntity.ok(existingUser);
    }
}
```

Request (PUT /api/users/update/1):

```
{
  "name": "Ajay Kumar",
  "role": "Full Stack Developer"
}
```

Response:

```
{
  "id": 1,
  "name": "Ajay Kumar",
  "role": "Full Stack Developer"
}
```

Q:

How It Works Internally

1. The DispatcherServlet scans and registers all controller methods with **@PutMapping**.

2. **When a PUT request comes to the mapped URL:**
   - Spring matches the request path (e.g., /update/{id}).
   - The @PathVariable extracts the id from the URL.
   - The @RequestBody deserializes the JSON body into a Java object (using Jackson).
   - The controller updates the resource and sends back a response (usually with HTTP 200 OK).

#. Example — Update with ResponseEntity

```
@PutMapping("/employees/{id}")
    public ResponseEntity<Employee> updateEmployee(@PathVariable int id, @RequestBody
Employee emp) {

   Employee existing = employeeService.findById(id);

   if (existing == null) {

      return ResponseEntity.notFound().build();
   }

   existing.setName(emp.getName());

   existing.setRole(emp.getRole());

   employeeService.save(existing);

   return ResponseEntity.ok(existing);
}
```

Response:

- **If updated → HTTP 200 OK**
- **If not found → HTTP 404 Not Found**

8. @DeleteMapping :

- **@DeleteMapping** is a composed annotation in Spring Boot used to handle HTTP DELETE requests.

#. **It's a shortcut for:**

@RequestMapping(method = RequestMethod.DELETE)

#. **Purpose:**

- It is used to delete a resource in database via Api (for example: deleting a user, removing an employee record, etc.)

#. **Basic Example :**

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

   private Map<Integer, User> users = new HashMap<>();

   @DeleteMapping("/delete/{id}")
    public ResponseEntity<String> deleteUser(@PathVariable int id) {
      User existingUser = users.remove(id);

      if (existingUser == null) {
            return ResponseEntity.status(HttpStatus.NOT_FOUND)
                .body("User not found with ID: " + id);
      }
      return ResponseEntity.ok("Deleted user with ID: " + id);
   }
}
```

Request:

DELETE http://localhost:8080/api/users/delete/1

Response:

Deleted user with ID: 1

If the user does not exist:

User not found with ID: 1

How It Works Internally ?

1. The DispatcherServlet scans for controller methods annotated with @DeleteMapping.

2. **When a DELETE request arrives:**
   - It checks the URL path and HTTP method.
   - Extracts the path variable (e.g., id).
   - Invokes the mapped controller method.
   - Returns a response with an appropriate HTTP status (200, 204, or 404).

9. @RequestBody :

- @RequestBody is used to bind the HTTP request body (usually in JSON format) to a Java object automatically.

- It is used in @PostMapping and @PutMapping

It tells Spring:

- "Take the incoming JSON (or XML) data from the request body and convert it into a Java object."

Example

- **Suppose you send this JSON request from Postman:**

```
{
 "id": 101,
 "name": "Ajay Kumar",
 "role": "Developer"
}
```

Controller

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

 @PostMapping("/add")
  public String addUser(@RequestBody User user) {
        return "User added: " + user.getName() + " (" + user.getRole() + ")";
  }
 }
```

Model Class (User.java)

```
public class User {
private int id;
private String name;
private String role;

// Getters and Setters
 }
```

What Happens Internally

1. The client sends JSON in the HTTP request body.

2. Spring Boot uses HttpMessageConverter (by default: Jackson library) to deserialize JSON → Java object.

3. The converted object is injected into the controller method parameter (User user).

4. You can directly use this Java object in your business logic.

#. Behind the Scenes — How It Works

1.**Spring Boot uses Jackson (a JSON processing library) through its MappingJackson2HttpMessageConverter to:**

- Convert JSON → Java object (for @RequestBody)
- Convert Java object → JSON (for @ResponseBody)

So when you write:

@PostMappingpublic ResponseEntity<User> createUser(@RequestBody User user)

Spring:

- Reads the JSON from request body.
- Maps it to User class fields using getters/setters.
- Passes the ready Java object to your method.

10 . @ResponseBody :

- **@ResponseBody tells Spring Boot that :**

1. the return value of a controller method should be written directly to the HTTP response body,
2. instead of rendering a view (like JSP or Thymeleaf).

#. In simple words:

- It converts Java objects into JSON (or XML) and sends it as the HTTP response body.
- It is commanly used with @GetMapping, @PostMapping, etc.

#. Example Without @ResponseBody

```
@Controller
public class UserController {

   @GetMapping("/user")
   public String getUser() {
      return "user";   // interpreted as a view name (like user.jsp or user.html)
   }
}
```

#. Example With @ResponseBody

```
@Controller
public class UserController {

   @GetMapping("/user")
   @ResponseBody
   public User getUser() {
      return new User(101, "Ajay Kumar", "Developer");
   }
}
```

Now Spring Boot:

1. Sees @ResponseBody

2. Converts the User object → JSON (using Jackson)

3. Sends it directly in the HTTP response

#. Response:

```
{
 "id": 101,
 "name": "Ajay Kumar",
 "role": "Developer"
}
```

#. Internal Working

@ResponseBody works with HttpMessageConverter (just like @RequestBody).

Flow:

1. Controller returns a Java object.

2. @ResponseBody signals Spring that the return value should not go to a view.

3. Spring uses MappingJackson2HttpMessageConverter to serialize Java → JSON.

4. JSON is sent back as the HTTP response body.

11 . @RequestParam :

- **@RequestParam** is used to extract query parameters from the URL or form data and bind them to method parameters in the controller.

- It's mostly used for optional inputs, filtering, or searching.

Example: Query Parameter in URL

Request URL:

GET http://localhost:8080/api/user?id=101&name=Ajay

Controller :

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

   @GetMapping("/user")
   public String getUserDetails(@RequestParam int id, @RequestParam String name) {
      return "User ID: " + id + ", Name: " + name;
   }
}
```

Output : User ID: 101, Name: Ajay

#. How It Works Internally

1. The client sends a GET or POST request with parameters (like ?id=101&name=Ajay).

2. The DispatcherServlet routes the request to the controller.

3. Spring automatically maps those parameter values to the variables in your method.

4. You can then use them directly in your logic.

#. Common Attributes of @RequestParam

Attribute	Description	Example
value / name	Parameter name in request	@RequestParam("id") int id
required	Whether parameter is mandatory (default: true)	@RequestParam(required = false)
defaultValue	Value to use if parameter is missing	@RequestParam(defaultValue = "Guest") String name

#. Example: Using All Attributes

```
@GetMapping("/welcome")
public String welcomeUser(
    @RequestParam(name = "name", required = false, defaultValue = "Guest") String userName) {
        return "Welcome, " + userName + "!";
}
```

Request:

http://localhost:8080/api/welcome?name=Ajay

Output: Welcome, Ajay!

Request (no param):

http://localhost:8080/api/welcome

Output: Welcome, Guest!

Example with Multiple Parameters :

```
@GetMapping("/search")
public String searchEmployee(
        @RequestParam String department,
        @RequestParam int experience) {
          return "Searching for employees in " + department + " with " + experience + " years of
experience.";
}
```

Request URL:

GET http://localhost:8080/api/search?department=IT&experience=3

Output :

Searching for employees in IT with 3 years of experience.

Q:

What is @PathVariable?

- **@PathVariable is used to extract values from the URL path and bind them to method parameters in your controller.**

- **It allows you to make dynamic RESTful URLs like:**

  /users/101
  /products/10
  /orders/5

Example: Simple Usage

Request URL:

GET http://localhost:8080/api/users/101

Controller:

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

  @GetMapping("/{id}")
   public String getUserById(@PathVariable int id) {
          return "Fetching user with ID: " + id;
   }
}
```

Response

Fetching user with ID: 101

Q: How It Works Internally :

1. Spring Boot reads the URL pattern from the @GetMapping (like /users/{id}).

2. When a request comes in (e.g., /users/101),

3. the value 101 is extracted and passed to the parameter int id.

4. You can use that variable to fetch, update, or delete the record from the database.

#. Common Attributes :

Attribute	Description	Example
value	The name of the path variable in the URL	@PathVariable("id")
required	Whether the parameter is mandatory (default: true)	@PathVariable(required = false)

#. Example — Custom Variable Name :

```
@GetMapping("/employee/{empId}")
public String getEmployee(@PathVariable("empId") int id) {
    return "Employee ID is: " + id;
}
```

URL: /employee/501

Output:

Employee ID is: 501

- Here, the URL placeholder {empId} is mapped to the parameter id using the ("empId") argument.

Example — Multiple Path Variables

```
@GetMapping("/department/{dept}/employee/{id}")
public String getEmployeeDetails(
        @PathVariable String dept,
        @PathVariable int id) {
    return "Department: " + dept + ", Employee ID: " + id;
}
```

Request URL: **GET http://localhost:8080/api/department/IT/employee/101**

Response: **Department: IT, Employee ID: 101**

Q:
What's Difference Between @PathVariable and @RequestParam ?

Feature	@PathVariable	@RequestParam
Source	Extracts values from the **URL path**	Extracts values from **query string or form**
URL Style	/users/101	/users?id=101
Use Case	Identify specific resource (like ID)	Filter, sort, or optional parameters
Commonly Used In	GET, PUT, DELETE	GET, POST
Example	/user/{id} → /user/101	/user?id=101

Real-Life Example — REST API Endpoint

```
@RestController
@RequestMapping("/api/employees")
public class EmployeeController {

    @GetMapping("/{id}")
    public ResponseEntity<Employee> getEmployeeById(@PathVariable int id) {
        Employee employee = employeeService.getEmployeeById(id);
        if (employee == null)
            return ResponseEntity.status(HttpStatus.NOT_FOUND).build();
        return ResponseEntity.ok(employee);
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<String> deleteEmployee(@PathVariable int id) {
        boolean deleted = employeeService.deleteEmployee(id);
        if (!deleted)
            return ResponseEntity.status(HttpStatus.NOT_FOUND)
                    .body("Employee not found with ID: " + id);
        return ResponseEntity.ok("Employee deleted successfully");
    }
}
```

✅ **Example URLs:**

- GET /api/employees/101 → Fetch employee
- DELETE /api/employees/101 → Delete employee

Q:

What is a Bean in Spring?

- In Spring Framework or Springboot , a **Bean** is simply an **object** that is **managed** by the **Spring IoC** (Inversion of Control) container.

- ✅ **A Bean** = **a Java object** that Spring creates, initializes, configures, and manages for you.

- You don't create it manually using new. Instead, Spring creates and injects it automatically when needed.

Example (Without Spring) : '

```
UserService userService = new UserService();    // You create the object manually
```

Example (With Spring) :

```
@Autowired
private UserService userService;                // Spring creates and injects the object
```

Note :

Here, UserService is a Spring Bean — managed by the container.

Q:

How Beans Are Created in Spring Boot

- There are two main ways to make Spring manage your class as a Bean:

By Using Stereotype Annotations

These are class-level annotations:

- **@Component**
- **@Service**
- **@Repository**
- **@Controller**
- **@RestController**

Example :

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

- Spring automatically scans it and creates a bean of this class.

Q:

What is @Bean?

- **@Bean is used to explicitly declare a bean in a Spring configuration class.**

- **It's typically used for:**
  1. Third-party library classes (that you can't annotate with @Component)

  2. Custom configuration of beans

Example 1:

```
@Configuration
public class AppConfig {

   @Bean
   public UserService userService() {
      return new UserService();       // 👈 Spring will manage this bean
   }
}
```

==> Now, anywhere in your project, you can inject it like this:

```
@Autowired
private UserService userService;
```

==> Spring knows about it because you defined it using @Bean.

Q: What Happens Internally ?

1. When Spring Boot starts, it scans all @Configuration classes.

2. It looks for methods annotated with @Bean.

3. It executes those methods once, creates an instance, and stores it in the ApplicationContext (Spring container).

4. When any other class needs that bean, Spring injects the same instance.

==> This is how Dependency Injection (DI) and Inversion of Control (IoC) work internally.

Example 2 : Using @Bean to Configure a Custom Object

- Let's say you need a custom object that's not part of your project classes — for example, a Password Encoder from Spring Security:

```
@Configuration
public class SecurityConfig {

   @Bean
   public PasswordEncoder passwordEncoder() {
      return new BCryptPasswordEncoder();       // Spring will manage this bean
   }
}
```

Then, anywhere in your app:

```
@Service
public class UserService {
   @Autowired
   private PasswordEncoder passwordEncoder;
   public void registerUser(String password) {
      System.out.println(passwordEncoder.encode(password));
   }
}
```

✅ **Output: Encrypted password (BCrypt)**

Q:
What is the default scope of a Spring Bean?

- 👉 The default bean scope in Spring is singleton.

🔹 Meaning of singleton scope

- Only one instance of the bean is created by the Spring container for the entire application context.

- Every time you inject or request that bean, you get the same shared instance.

🔹 Example

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

@Component
public class SweetService {

   public SweetService() {

      System.out.println("SweetService bean created!");
   }
}
```

== > Even without using @Scope, the bean is singleton by default.

Test:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class TestBean {

   @Autowired
   private SweetService sweetService1;

   @Autowired
   private SweetService sweetService2;

   public void testScope() {
      System.out.println(sweetService1 == sweetService2); // true ✅
   }
}
```

✅ **Output:**

```
SweetService bean created!
true
```

👉 Only one object is created — both references point to the same instance.

#. Other Bean Scopes in Spring :

Scope	Description
singleton	(Default) Only one instance per Spring container.
prototype	A new instance every time it's requested.
request	One instance per HTTP request (web apps only).
session	One instance per HTTP session.
application	One instance per ServletContext.
websocket	One instance per WebSocket session.

#. 🔹 Example (Change Scope) :

```
@Component
@Scope("prototype")
public class SweetService {

   public SweetService() {

      System.out.println("New SweetService bean created!");
   }
}
```

==> Now every time you @Autowired or getBean() it, a new object is created.

Q: What is @Scope in Spring?

- @Scope is a Spring annotation used to define the lifecycle and visibility (scope) of a Spring bean.

- It tells the Spring container how many instances of a bean should be created and how long they should live.

🔹 Basic Syntax

```
@Scope("scope-name")
```

==> You can use it with annotations like @Component, @Service, @Repository, etc.

Q:
What's differences between @Bean and @Component ?

Feature	@Bean	@Component
Where Used	On a **method**	On a **class**
When Used	When you can't modify the class (3rd-party, library)	For your own components/services
Created By	A method in @Configuration class	Automatically via component scan
Control	Full control over bean creation	Default, auto-detected

Example — Multiple Beans of Same Type :

```
@Configuration
public class PaymentConfig {

    @Bean
    public PaymentService paypalService() {
        return new PaypalService();
    }

    @Bean
    public PaymentService stripeService() {
        return new StripeService();
    }
}
```

==> Now, to inject a specific one:

```
@Autowired
@Qualifier("stripeService")
private PaymentService paymentService;
```

==> ✅ Both beans are created by Spring and available in the context.

Q: Explain about Bean Lifecycle (High-Level) ?

==>Each Bean in Spring goes through these phases:

1. **Instantiation** – Spring creates the object.
2. **Dependency Injection** – Injects required dependencies.
3. **Initialization** – Calls methods like @PostConstruct.
4. **Usage** – Bean is ready for use.
5. **Destruction** – On shutdown, Spring destroys the bean properly.

Q:

What is IoC (Inversion of Control)?

- Inversion of Control (IoC) means giving control of object creation and dependency management to the Spring Framework, instead of you doing it manually in your code.
- **Normally (in Java), we create and manage objects like this:**

```
UserService service = new UserService();
OrderService order = new OrderService(service);
```

- Here, you control how and when objects are created — that's tight coupling.
- But in Spring, you just declare what dependencies you need, and Spring injects them automatically:

```
@Autowired
private UserService userService;
```

***. Here, Spring creates and injects UserService into OrderService.**

***. This concept of Spring controlling object creation and dependency wiring = Inversion of Control (IoC).**

Q: What is the IoC Container?

- The IoC Container is the core part of the Spring Framework that manages the lifecycle of beans —

It is responsible for:

1. **Creating objects (beans)**
2. **Managing dependencies (Dependency Injection)**
3. **Configuring them (from XML, annotations, or Java config)**
4. **Managing their lifecycle (init → destroy)**

Q: How IoC Works Internally (Step by Step) :

1. Application starts (SpringApplication.run() is called).
2. Spring Boot creates an ApplicationContext (IoC Container).
3. It scans your project for beans (via component scanning).
4. It creates and configures all beans.
5. When a bean needs another bean (via @Autowired), Spring injects it.
6. When the app stops, it destroys beans gracefully.

#. Types of IOC Containers :

- Spring provides two main types of IoC containers:

1. BeanFactory

2. ApplicationContext

BeanFactory (Basic Container) :

- BeanFactory is the simplest and oldest IoC container provided by Spring.
- It provides basic dependency injection support.
- **Example:**

```
Resource resource = new ClassPathResource("beans.xml");

BeanFactory factory = new XmlBeanFactory(resource);

UserService userService = (UserService) factory.getBean("userService");
```

#. Key Characteristics:

Feature	Description
Lazy Loading	Beans are created **only when requested** (on-demand).
Lightweight	Very minimal memory footprint.
No advanced features	Doesn't support Spring AOP, events, or internationalization.
Usage	Rarely used in modern Spring applications.

⚠️ Why BeanFactory is Rarely Used Now:

==> Because it's too basic — it doesn't support many enterprise-level features like:

- Application events
- Automatic bean post-processing
- Message sources (i18n)
- Web context features

==> That's where ApplicationContext comes in.

ApplicationContext (Advanced Container)

- ApplicationContext is the enhanced version of BeanFactory.
- It includes all BeanFactory features + enterprise-level functionalities.
- It's the default IoC container in Spring and Spring Boot.

Example:

```
ApplicationContext context =
        new ClassPathXmlApplicationContext("beans.xml");

UserService userService = context.getBean(UserService.class);
```

✅ Features of ApplicationContext:

Feature	Description
Eager Loading	All singleton beans are created at startup (for better performance).
Event Handling	Supports application events (e.g., ContextRefreshedEvent).
AOP Support	Integrates with Aspect-Oriented Programming.
Internationalization (i18n)	Supports message sources for multiple languages.
Annotation Support	Supports annotations like @Autowired, @Component, etc.
Profiles	Allows environment-based configuration (e.g., dev/test/prod).
Web Integration	Works with web-aware contexts (WebApplicationContext).

#. Types of ApplicationContext Implementations :

Implementation	Description
ClassPathXmlApplicationContext	Loads configuration from XML file located in classpath.
FileSystemXmlApplicationContext	Loads configuration from an XML file located in the file system.
AnnotationConfigApplicationContext	Loads configuration from Java-based @Configuration classes.
WebApplicationContext	Used in web applications — integrates with Spring MVC and Servlets.

#. Example (Modern Spring Boot Way) :

- **In Spring Boot, you don't need to create the container manually — it's done automatically when you call:**

```
@SpringBootApplication
public class DemoApplication {

   public static void main(String[] args) {

      ApplicationContext context =
         SpringApplication.run(DemoApplication.class, args);

      UserService userService = context.getBean(UserService.class);

      System.out.println(userService.getUser());
   }
}
```

Here:

- SpringApplication.run() → creates the ApplicationContext.
- It automatically scans, creates, and wires all beans.

✅ So, Spring Boot internally uses ApplicationContext as its IoC container.

Q:

What is @Autowired?

- **@Autowired** is used for **dependency injection** in Spring. It allows Spring to automatically detect and inject dependent **beans** (classes or objects) dependencies into another class **without manually creating them using new.**

Example :

```
@Service
public class UserService {

   @Autowired
   private UserRepository userRepository;

   public List<User> getAllUsers() {

      return userRepository.findAll();

   }
}
```

Here:

- UserRepository is a Spring-managed bean
- Spring automatically creates and injects its instance into UserService
- You don't need to write **userRepository = new UserRepositoryImpl()** manually.

Q: How It Works Internally ?

When Spring Boot starts:

1. The Spring container scans all packages (using @ComponentScan).
2. It finds all beans annotated with @Component, @Service, @Repository, @Controller, etc.
3. It checks where @Autowired is used.
4. It matches the dependency by type (class/interface) and injects the right bean automatically.

==>So, it's Spring's way of doing Inversion of Control (IoC) — the framework controls object creation.

Q:

How can achive @Autowired ?

- **Spring allows @Autowired in 3 ways :**

  Using Constructor, Setter, and Field Injection .

Field Injection (most common but less testable) :

```
@Autowired
private UserRepository userRepository;
```

Note :

- It is Simple to use but ⚠️ Harder to test (no constructor)

Constructor Injection (✅ Recommended best practice) :

```
@Service
public class UserService {
   private final UserRepository userRepository;

   @Autowired    // optional in Spring 4.3+ (auto-detected for single constructor)
   public UserService(UserRepository userRepository) {
      this.userRepository = userRepository;
   }
}
```

✅ **Advantages:**

- Works with final fields
- Easy to test
- Detects missing dependencies at compile time

Setter Injection :

```
@Autowired
public void setUserRepository(UserRepository userRepository) {
this.userRepository = userRepository;
}
```

==> Useful when the dependency is optional or can change later.

Q: What Happens If There Are Multiple Beans?

- If Spring finds more than one bean of the same type → it gets confused.
- And Spring throws **NoUniqueBeanDefinitionException.**
- To resolve this, we can use **@Qualifier** to specify which bean to inject or mark one bean as **@Primary** to make it the default."

#. First Let's see

Q: How Error will occur using Real - case Example :

⚠️ Example (Error Case)

```
public interface PaymentService {

   void pay();
}

@Service
public class PaypalService implements PaymentService {

   public void pay() {

      System.out.println("Payment done through PayPal");
   }
}

@Service
public class StripeService implements PaymentService {

   public void pay() {

      System.out.println("Payment done through Stripe");
   }
}

@Service
public class PaymentProcessor {

   @Autowired
   private PaymentService paymentService;      // ❌ Error: Two beans found
}
```

Note :

1. **W**hen Spring starts, it finds two beans implementing PaymentService:
   - paypalService
   - stripeService

2. Spring is confused — which one to inject?

3. So it throws the NoUniqueBeanDefinitionException.

✅ **Solution 1: Use @Qualifier**

- **You can explicitly tell Spring which bean to inject by name.**

Example :

```
@Service
public class PaymentProcessor {

    @Autowired
    @Qualifier("stripeService")      // 👈 Choose specific bean name
    private PaymentService paymentService;

    public void processPayment() {
        paymentService.pay();
    }
}
```

Explanation:

- Bean names default to the **class name with lowercase first letter**, unless overridden:
  - PaypalService → paypalService
  - StripeService → stripeService

✅ **So here Spring injects StripeService only.**

✅ **Solution 2: Use @Primary**

- **If you want to set a default bean, mark it as @Primary.**

Example :

```
@Service
@Primary    // 👈 Make this default bean
public class PaypalService implements PaymentService {
    public void pay() {
        System.out.println("Payment done through PayPal (default)");
    }
}
@Service
public class StripeService implements PaymentService {
    public void pay() {
        System.out.println("Payment done through Stripe");
    }
}
@Service
public class PaymentProcessor {

    @Autowired
    private PaymentService paymentService;   // ✅ Automatically injects PaypalService
}
```

- **Now, Spring always injects PaypalService by default,**
- **unless you explicitly use @Qualifier("stripeService").**

✅ Solution 3: Use Both Together

- If you want a default bean but sometimes need to switch manually,
- you can combine both @Primary and @Qualifier.

Example:

```
@Service
@Primary
public class PaypalService implements PaymentService { ... }

@Service
public class StripeService implements PaymentService { ... }

@Service
public class PaymentProcessor {

   @Autowired
   @Qualifier("stripeService")
   private PaymentService paymentService;  // Inject Stripe instead of default(Overrides  @Primary )
}
```

Note :

- Even though PaypalService is @Primary,
- Spring will inject StripeService because @Qualifier has **higher priority.**

Q :

What is @Qualifier?

- @Qualifier is used along with @Autowired to specify which bean should be injected when multiple beans of the same type exist in the Spring container.
- It helps Spring resolve ambiguity during dependency injection.

Q: How It Works Internally

- Spring first finds all beans of type PaymentService.
- Since there are multiple, it checks the @Qualifier value.
- It injects the bean whose name matches the qualifier name.

In our case:

- Bean name = stripeService
- Qualifier value = "stripeService"
- ✅ → Match found → Injected successfully.

Q:
Where You Can Use @Qualifier ?

On Fields :

```
@Autowired
@Qualifier("paypalService")
private PaymentService paymentService;
```

On Constructors

```
@Autowired
public PaymentProcessor(@Qualifier("stripeService") PaymentService paymentService) {
    this.paymentService = paymentService;
}
```

On Setter Methods

```
@Autowired
public void setPaymentService(@Qualifier("stripeService") PaymentService paymentService) {
    this.paymentService = paymentService;
}
```

Q:
What is @Primary?

- "@Primary is used when multiple beans of the same type exist. It tells Spring to inject that bean by default whenever ambiguity arises during dependency injection.
- It's often used when one bean should be preferred across the application."

Then add:

- "If we need a different bean in a specific class, we can override @Primary by using @Qualifier."

✅ Summary Table :

Annotation	Purpose	Priority
@Autowired	Auto-wires by type	—
@Qualifier("beanName")	Specifies exact bean	🎯 Highest
@Primary	Sets default bean globally	🥈 Lower than @Qualifier

Q: What is DTO ?

- DTO (Data Transfer Object) is a **simple Java object** used to transfer data between layers (like Controller → Service → Client).

- It's not an Entity — it's used only for data exchange.

Note : In simple word's

- It's just a simple Java class that is used to carry data from one place to another — especially between the backend and the frontend (or between different layers in your app).

. Example Scenario

- You have a User Entity that maps to your database table:

```
@Entity
public class User {

   @Id
   private Long id;
   private String username;
   private String password;
   private String email;
   private String role;

}
```

- **==> But when returning data to the frontend, you don't want to expose sensitive fields like password.**

==> So, you create a UserDTO:

```
public class UserDTO {

private String username;
private String email;

}
```

==> Then, in your code, you convert :

```
User user = userRepository.findById(1L).get();
UserDTO dto = new UserDTO(user.getUsername(), user.getEmail());
 return dto;
```

==> The DTO hides unnecessary/sensitive data and simplifies API responses.

Q: Why Use DTOs?

Reason	Explanation
1. Security	Prevents exposing sensitive info (like passwords, tokens)
2. Loose Coupling	Decouples database entities from API responses
3. API Control	You can customize what you send/receive in APIs
4. Validation	DTOs can have @Valid annotations for request input
5. Clean Architecture	Separates persistence layer (Entity) and web layer (Controller)

#. Example: DTO in REST API (Spring Boot)

Step 1: Entity

```
@Entity
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String username;
    private String password;
    private String email;
}
```

Step 2: DTO Class

```
public class UserDTO {

    private String username;
    private String email;

}
```

Step 3: Controller (using DTO)

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

    @Autowired
    private UserRepository repository;

    @GetMapping("/{id}")
    public UserDTO getUser(@PathVariable Long id) {

        User user = repository.findById(id)
                .orElseThrow(() -> new RuntimeException("User not found"));

        UserDTO dto = new UserDTO();
        dto.setUsername(user.getUsername());
        dto.setEmail(user.getEmail());
        return dto;
    }
}
```

==> Controller returns DTO, not the full entity → hides sensitive info.

#. Using DTO for Input (Request)

- **Sometimes we use DTOs for input validation too:**

#. Example: UserRequestDTO

```
public class UserRequestDTO {
    @NotBlank
    private String username;

    @NotBlank
    private String password;

    @Email
    private String email;
}
```

#. And in Controller:

```
@PostMapping("/register")
  public String registerUser(@Valid @RequestBody UserRequestDTO dto) {

   User user = new User();

   user.setUsername(dto.getUsername());
   user.setPassword(passwordEncoder.encode(dto.getPassword()));
   user.setEmail(dto.getEmail());

   userRepository.save(user);
       return "User registered!";
}
```

==> Clean, validated, and prevents database misuse.

#. Tools to Convert Entity ↔ DTO

==> Manually converting works fine, but for larger projects:

- ModelMapper
- MapStruct

#. Example (ModelMapper):

```
        ModelMapper mapper = new ModelMapper();
        UserDTO dto = mapper.map(user, UserDTO.class);
```

#. Real-World Example (Spring Boot + REST API)

In professional projects:

- Entities map to the database
- DTOs are used in Controllers for API input/output
- Mappers convert between them automatically

Q: How It Works (Simple Flow) :

==> Here's how a DTO fits in a Spring Boot REST API:

Database (map to Entity)
↓
Repository
↓
Service
↓
Controller
↓
DTO (Sent to frontend)

==> So, the Entity is connected to the database,
==> and the DTO is connected to the API (frontend).

. Common Validation Annotations in DTOs

Annotation	Description	Example
@NotNull	Field must not be null	Integer id
@NotBlank	String must not be blank or empty	Name
@NotEmpty	Field must not be empty (for lists, sets, strings)	List roles
@Email	Must be a valid email	Email
@Size(min, max)	Must be within length range	Password
@Pattern	Must match regex	Phone number
@Min / @Max	Numeric range validation	Age

1. @NotBlank :

- Ensures the field is not null, not empty, and not only spaces.

#. Example :

```
 import jakarta.validation.constraints.NotBlank;
public class UserDTO {

   @NotBlank(message = "Name cannot be blank")
   private String name;

   @NotBlank(message = "Email cannot be blank")
   private String email;
}
```

❌ Invalid: **Here, if someone sends:**

```
{
    "name": " ",
     "email": "ajay@gmail.com"
}
```

==> Spring Boot will automatically respond with a 400 Bad Request and a message like:

Valid:

```
{
   "name": "Ajay","
   email": "ajay@gmail.com"
}
```

2. @Email :

- Ensures the value is a valid email format (contains @ and domain).

❌ **Invalid:**

```
{
  "email": "ajay"
}
```

✅ **Valid:**

```
{
  "email": "ajay@gmail.com"
}
```

Example :

```
import jakarta.validation.constraints.Email;
import jakarta.validation.constraints.NotBlank;

public class UserDTO {
    @NotBlank(message = "Name is required")
    private String name;

    @Email(message = "Email should be valid")
    @NotBlank(message = "Email is required")
    private String email;
}
```

✅ This checks both:

1. The email field is not blank
2. The email format is valid

#. Using Validation in Controller

==> You must use @Valid in your controller to activate validation:

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

    @PostMapping
    public ResponseEntity<String> createUser(@Valid @RequestBody UserDTO user) {
        return ResponseEntity.ok("User created: " + user.getName());
    }
}
```

==> If invalid data is sent, Spring Boot automatically handles it with an error response like:

```
{
 "timestamp": "2025-10-05T08:30:00",
 "status": 400,
 "errors": [
  "Name is required",
  "Email should be valid"
 ]
}
```

3. @NotNull :

- It ensures that the field must not be null.

==> ✅ But it does not check if the field is empty or whitespace.
That means:

- **@NotNull** allows "" (empty string)
- But rejects null

Example : DTO

```
import jakarta.validation.constraints.NotNull;
public class UserDTO {

  @NotNull(message = "Name cannot be null")
  private String name;
}
```

4. @NotEmpty :

1. The field is not null.
2. The field is not empty.

✅ It applies to:

- Strings
- Collections (List, Set, etc.)
- Arrays

==> But it does allow whitespace-only strings (unlike @NotBlank).

Example :

```
@NotEmpty(message = "Name cannot be empty")
 private String name;
```

Q: Difference Between @NotNull, @NotEmpty, and @NotBlank ?

Annotation	Null Allowed?	Empty Allowed?	Whitespaces Allowed?
@NotNull	❌ No	✅ Yes	✅ Yes
@NotEmpty	❌ No	❌ No	✅ Yes
@NotBlank	❌ No	❌ No	❌ No

5. @Size(min, max) :

- **@Size** is a validation annotation (from jakarta.validation.constraints) used to specify the minimum and maximum allowed size of a field.

==> It works with:

- String (length)
- Collections (number of elements)
- Arrays
- Maps

#. Example :

```
@Size(min = 3, max = 10, message = "Name must be between 3 and 10 characters")
 private String name;
```

This means:

- Length must be ≥ 3 and ≤ 10
- If shorter than 3 or longer than 10 → ❌ validation fails

6. @Pattern :

- It is used to validate that a string field matches a specific regular expression (regex).

Basic Syntax :

```
@Pattern(regexp = "your-regex", message = "custom message")
```

- It only works on String fields.
- If the value doesn't match the regex, validation will fail ❌.

#. Simple Example: Allow only Alphabets

```
@Pattern(regexp = "^[A-Za-z]+$", message = "Name must contain only letters")
private String name;
```

✅ Valid inputs:

- "Ajay"
- "Kumar"

❌ Invalid inputs:

- "Ajay123"
- "Ajay@"
- "Ajay Kumar" (contains space)

7. @Min / @Max :

- It is used to specify numeric limits (minimum and maximum values) for numbers (like int, long, double, etc.).

#. Basic Syntax :

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

@Max(value = 60, message = "Age must be less than or equal to 60")
private int age;
```

#. Meaning

- @Min(18) → value must be >= 18
- @Max(60) → value must be <= 60

==> If the value is outside the range, validation fails ❌

#. ❌ Not Supported For:

- Strings
- Collections
- Objects

==> (because it's meant for numbers only)

8. @Valid :

- @Valid is a Jakarta validation annotation (from jakarta.validation.Valid) used to enable validation of an object — typically in a Spring Boot controller, service, or nested object.

- **It tells Spring:**
  "Hey, before using this object, check all the validation rules defined inside it."

- **@Valid** is used With **@RequestBody** in controllers

In Simple Words:

- @Valid is the trigger that activates all the validation annotations inside your DTO (like @NotNull, @Size, @Email, etc.)

- Without @Valid, the validations won't run ❌ — even if you wrote all the annotations correctly.

Example: Using @Valid in Controller

- **Let's say you have a DTO:**

```
import jakarta.validation.constraints.*;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class UserDTO {

    @NotBlank(message = "Name is required")
    @Size(min = 3, max = 15, message = "Name must be between 3 and 15 characters")
    private String name;

    @Email(message = "Invalid email format")
    private String email;

    @Min(value = 18, message = "Age must be at least 18")
    private int age;
}
```

#. Controller without @Valid ❌:

```
@PostMapping("/add")
public ResponseEntity<String> addUser(@RequestBody UserDTO user) {
    return ResponseEntity.ok("User added successfully!");
}
```

- 👉 Even if you send invalid data (like blank name, invalid email),
- Spring will NOT validate the request — it'll just pass the object as-is.

#. Controller with @Valid :

```
    @PostMapping("/add")
    public ResponseEntity<String> addUser(@Valid @RequestBody UserDTO user) {
      return ResponseEntity.ok("User added successfully!");
  }
```

==> Now, @Valid activates validation for all annotations inside UserDTO.

#. ❌ Invalid Request Example :

```
{
 "name": "",
 "email": "ajaygmail.com",
 "age": 15
}
```

#. Response:

```
{
 "timestamp": "2025-10-05T11:00:00",
 "status": 400,
 "errors": [
  "Name is required",
  "Invalid email format",
  "Age must be at least 18"
 ]
}
```

#. Behind the Scenes (How It Works)

1. When you annotate a request object with @Valid,
2. Spring Boot calls the Bean Validation API (Jakarta Validation).
3. It checks each field of your object for:
    - @NotNull
    - @NotEmpty
    - @Size
    - @Email
    - @Pattern
    - @Min
    - @Max
    - ...and so on.
4. If any rule fails → Spring throws **MethodArgumentNotValidException.**
5. Spring automatically returns a 400 Bad Request with the validation errors.

Q:
What is a REST API ?

==> REST (Representational State Transfer) is an architectural style for designing web services that use HTTP to communicate between client and server.

- Client → sends an HTTP request (GET, POST, PUT, DELETE)
- Server → returns an HTTP response, usually in JSON format.

#. Key REST Concepts :

Concept	Description
Resource	Data entity (e.g., /users, /products)
URI (Uniform Resource Identifier)	Path to access a resource (/api/users/1)
HTTP Methods	Define action on the resource (GET, POST, PUT, DELETE)
Status Codes	Represent the result of an operation (200 OK, 404 Not Found, 201 Created, etc.)
JSON/XML	Common formats for data exchange (mostly JSON now)

#. Building REST APIs :

Best Practices

- **Use DTOs** (Data Transfer Objects) for request/response instead of entities.
- **Add Exception Handling** using @ControllerAdvice.
- **Use Validation** (@Valid, @NotBlank, etc.).
- **Follow REST naming conventions.**
- **Add Swagger for API documentation** (springdoc-openapi-ui dependency).

- **Let's now build a complete industry-level REST API project in Spring Boot — step by step — with all professional standards like:**

- **DTOs + Validation + Exception Handling + Swagger UI + Layered Architecture**

- **We'll build a Student Management REST API (clean, production-ready).**

⚙️ Step 1: Project Setup

👉 Go to https://start.spring.io

Select:

- Project: Maven
- Language: Java
- Spring Boot: 3.x+
- Dependencies:
  - Spring Web
  - Spring Data JPA
  - MySQL Driver
  - Lombok
  - Validation
  - Springdoc OpenAPI (for Swagger)

== > Click Generate, extract the ZIP, and open it in IntelliJ / Eclipse / VS Code.

⚙️ Step 2: Configure application.properties

```
# Database Configuration
spring.datasource.url=jdbc:mysql://localhost:3306/studentdb?useSSL=false
spring.datasource.username=root
spring.datasource.password=root

# JPA Config
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true

# Swagger Config
springdoc.api-docs.path=/api-docs
springdoc.swagger-ui.path=/swagger-ui.html
```

Step 3: Entity Layer

```
package com.example.studentmanagement.entity;
import jakarta.persistence.*;
import lombok.*;

@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
@Table(name = "students")
public class Student {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String email;
    private String department;
}
```

Step 4: DTO Layer (Best Practice)

- DTO helps decouple entity from API data format.

```
package com.example.studentmanagement.dto;
import jakarta.validation.constraints.Email;
import jakarta.validation.constraints.NotBlank;
import lombok.*;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class StudentDTO {

    private Long id;

    @NotBlank(message = "Name cannot be empty")
    private String name;

    @Email(message = "Email should be valid")
    @NotBlank(message = "Email cannot be empty")
    private String email;

    @NotBlank(message = "Department cannot be empty")
    private String department;
}
```

Step 5: Repository Layer

```
package com.example.studentmanagement.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import com.example.studentmanagement.entity.Student;

public interface StudentRepository extends JpaRepository<Student, Long> {

}
```

Step 6: Service Layer

```
package com.example.studentmanagement.service;

import com.example.studentmanagement.dto.StudentDTO;
import java.util.List;

public interface StudentService {

    List<StudentDTO> getAllStudents();

    StudentDTO getStudentById(Long id);

    StudentDTO createStudent(StudentDTO studentDTO);

    StudentDTO updateStudent(Long id, StudentDTO studentDTO);

    void deleteStudent(Long id);
}
```

Implementation :

```
package com.example.studentmanagement.service.impl;

import com.example.studentmanagement.dto.StudentDTO;
import com.example.studentmanagement.entity.Student;
import com.example.studentmanagement.exception.ResourceNotFoundException;
import com.example.studentmanagement.repository.StudentRepository;
import com.example.studentmanagement.service.StudentService;
import org.springframework.stereotype.Service;
import java.util.List;
import java.util.stream.Collectors;

@Service
public class StudentServiceImpl implements StudentService {

    private final StudentRepository repository;

    public StudentServiceImpl(StudentRepository repository) {
        this.repository = repository;
    }

    private StudentDTO convertToDTO(Student student) {
        return new StudentDTO(student.getId(), student.getName(), student.getEmail(),
student.getDepartment());
    }

    private Student convertToEntity(StudentDTO dto) {
        return new Student(dto.getId(), dto.getName(), dto.getEmail(), dto.getDepartment());
    }

    @Override
    public List<StudentDTO> getAllStudents() {
        return repository.findAll().stream().map(this::convertToDTO).collect(Collectors.toList());
    }

    @Override
    public StudentDTO getStudentById(Long id) {
        Student student = repository.findById(id)
            .orElseThrow(() -> new ResourceNotFoundException("Student not found with ID: " + id));
        return convertToDTO(student);
    }
```

```
@Override
 public StudentDTO createStudent(StudentDTO dto) {

 Student student = convertToEntity(dto);

       return convertToDTO(repository.save(student));
 }

 @Override
 public StudentDTO updateStudent(Long id, StudentDTO dto) {

 Student student = repository.findById(id)
        .orElseThrow(() -> new ResourceNotFoundException("Student not found with ID: " + id));

 student.setName(dto.getName());

 student.setEmail(dto.getEmail());

 student.setDepartment(dto.getDepartment());

 return convertToDTO(repository.save(student));
 }

 @Override
 public void deleteStudent(Long id) {

    if (!repository.existsById(id)) {
                        throw new ResourceNotFoundException("Student not found with ID: " + id);
    }
      repository.deleteById(id);
  }
}
```

⚠️ Step 7: Custom Exception Handling

```
package com.example.studentmanagement.exception;

public class ResourceNotFoundException extends RuntimeException {

   public ResourceNotFoundException(String message) {

       super(message);
   }
}
```

#. Global Exception Handler :

```
package com.example.studentmanagement.exception;

import org.springframework.http.*;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.context.request.WebRequest;
import java.time.LocalDateTime;
import java.util.HashMap;
import java.util.Map;

@RestControllerAdvice
public class GlobalExceptionHandler {

   @ExceptionHandler(ResourceNotFoundException.class)
      public ResponseEntity<Object> handleResourceNotFound(ResourceNotFoundException ex ,
WebRequest request) {

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

       body.put("timestamp", LocalDateTime.now());

       body.put("message", ex.getMessage());

       body.put("status", HttpStatus.NOT_FOUND.value());

       return new ResponseEntity<>(body, HttpStatus.NOT_FOUND);
   }

   @ExceptionHandler(Exception.class)
      public ResponseEntity<Object> handleGeneral(Exception ex, WebRequest request) {

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

       body.put("timestamp", LocalDateTime.now());

       body.put("message", ex.getMessage());

       body.put("status", HttpStatus.INTERNAL_SERVER_ERROR.value());

       return new ResponseEntity<>(body, HttpStatus.INTERNAL_SERVER_ERROR);
   }
}
```

Step 8: Controller Layer

```
package com.example.studentmanagement.controller;
import com.example.studentmanagement.dto.StudentDTO;
import com.example.studentmanagement.service.StudentService;
import jakarta.validation.Valid;
import org.springframework.http.*;
import org.springframework.web.bind.annotation.*;
import java.util.List;

@RestController
@RequestMapping("/api/students")
public class StudentController {

    private final StudentService service;

    public StudentController(StudentService service) {
        this.service = service;
    }

    @GetMapping
    public ResponseEntity<List<StudentDTO>> getAll() {
            return ResponseEntity.ok(service.getAllStudents());
    }

    @GetMapping("/{id}")
    public ResponseEntity<StudentDTO> getById(@PathVariable Long id) {
            return ResponseEntity.ok(service.getStudentById(id));
    }

    @PostMapping
    public ResponseEntity<StudentDTO> create(@Valid @RequestBody StudentDTO dto) {
         return new ResponseEntity<>(service.createStudent(dto), HttpStatus.CREATED);
    }

    @PutMapping("/{id}")
    public ResponseEntity<StudentDTO> update(@PathVariable Long id, @Valid @RequestBody
StudentDTO dto) {
            return ResponseEntity.ok(service.updateStudent(id, dto));
    }
    @DeleteMapping("/{id}")
    public ResponseEntity<Void> delete(@PathVariable Long id) {
        service.deleteStudent(id);
            return ResponseEntity.noContent().build();
    }
}
```

Step 9: Test Endpoints in Postman

Method	URL	Description
GET	**/api/students**	Get all students
GET	**/api/students/{id}**	Get student by ID
POST	**/api/students**	Create student
PUT	**/api/students/{id}**	Update student
DELETE	**/api/students/{id}**	Delete student

Step 10: API Documentation with Swagger

After running the project →

Open: http://localhost:8080/swagger-ui.html

==> You'll get an interactive Swagger UI for testing APIs 👨‍💻

Output Example (POST → Create Student)

Request:

```
{
  "name": "Ajay Kumar",
  "email": "ajay@gmail.com",
  "department": "Computer Science"
}
```

Response:

```
{
  "id": 1,
  "name": "Ajay Kumar",
  "email": "ajay@gmail.com",
  "department": "Computer Science"
}
```

Q:

What is @Data in Java (Lombok)?

==> @Data is a Lombok annotation that automatically generates boilerplate code for a class like:

1. Getters
2. Setters
3. toString()
4. equals() and hashCode()
5. Required constructor (@RequiredArgsConstructor)

==> So, you don't have to write them manually.

#. Example Without Lombok :

```
public class User {
    private String name;
    private String email;

    // Getters and Setters
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public String getEmail() { return email; }
    public void setEmail(String email) { this.email = email; }

    // toString
    @Override
    public String toString() {
        return "User{name='" + name + "', email='" + email + "'}";
    }
}
```

==> This is a lot of code — repetitive and boring.

#. Example With Lombok @Data :

```
import lombok.Data;

@Data
public class User {
    private String name;
    private String email;
}
```

That's it! 💥

Lombok will auto-generate all getters, setters, toString(), etc. during compilation.

⚠️ **Important Note :**

==> Don't use @Data on entities if you have complex relationships (like OneToMany, ManyToOne).

Instead, use:

@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor

Q:

What is @NoArgsConstructor?

- **@NoArgsConstructor** is a Lombok annotation that automatically creates a no-argument (default) constructor for your class.

- ✅ It's equivalent to writing:

```
public User() {
}
```

#. Example Without Lombok :

```
public class User {
   private String name;
   private String email;

   public User() {
       // Default constructor auto generate
   }
}
```

#. Example With Lombok

```
import lombok.NoArgsConstructor;

@NoArgsConstructor
public class User {
   private String name;
   private String email;
}
```

==> 💥 Lombok automatically generates the empty constructor during compilation.

==> so if you will used @Data then default constructor will not provide by java compiler you.

Q: Why Is @NoArgsConstructor Important?

1. Required by JPA / Hibernate

- When you mark a class as an @Entity, Hibernate needs a no-argument constructor (at least protected or public).

- Because when Hibernate loads data from the database, it:
    1. Creates an empty object
    2. Then sets fields using reflection

==> So every Entity class must have a default constructor.

#. Example :

```
@Entity
@NoArgsConstructor
@AllArgsConstructor
public class User {

   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
   private String name;
}
```

==> Without @NoArgsConstructor, Hibernate will throw: No default constructor for entity ❌

Q: What is @AllArgsConstructor?

- **@AllArgsConstructor** is a Lombok annotation that automatically generates a constructor with all class fields as parameters.

#. Example Without Lombok

```
public class User {
   private Long id;
   private String name;
   private String email;

   public User(Long id, String name, String email) {
      this.id = id;
      this.name = name;
      this.email = email;
   }
}
```

#. Example With Lombok :

```
import lombok.AllArgsConstructor;
@AllArgsConstructor
public class User {
   private Long id;
   private String name;
   private String email;
}
```

==> 💥 Lombok automatically generates this constructor during compilation:

```
public User(Long id, String name, String email) {
   this.id = id;
   this.name = name;
   this.email = email;
}
```

#. Real-World Use in Spring Boot

#. Example: DTO Mapping in Service

```
User user = new User(1L, "Ajay", "ajay@gmail.com");
UserDTO dto = new UserDTO(user.getName(), user.getEmail());
```

==> So @AllArgsConstructor helps you easily pass data around without writing boilerplate constructors manually.

 # . Common Lombok Combination

Annotation	Purpose
@Data	Generates getters, setters, toString, equals, hashCode
@NoArgsConstructor	Adds empty constructor
@AllArgsConstructor	Adds full-field constructor

⚠️ Important Tips :

- Use **@NoArgsConstructor** for Hibernate and JSON serialization (needs empty constructor).
- Use **@AllArgsConstructor** when you want to easily create objects or use DTO mapping.
- Together, they make your class flexible for both frameworks and manual usage.

Q:
What is @RestControllerAdvice?

- @RestControllerAdvice is a special annotation in Spring Boot used to handle exceptions globally across all REST controllers.

- It's like saying:
  "Hey Spring, if any controller in my app throws an exception — don't crash, just send a clean JSON response instead."

#. In Simple Words:

- It's a global error handler for your REST API.

- Instead of writing try-catch blocks in every controller , you write one centralized class to handle all exceptions.

Q: How It Works :

1. You create a class and annotate it with @RestControllerAdvice.

2. Inside it, you define methods annotated with @ExceptionHandler(ExceptionType.class).

3. Whenever an exception occurs in any controller, Spring will automatically call the matching @ExceptionHandler method.

#. Example: Global Exception Handler

==> Let's say you have validation in your controller like this:

```
@PostMapping("/users")
public ResponseEntity<String> addUser(@Valid @RequestBody UserDTO user) {
    return ResponseEntity.ok("User added successfully!");
}
```

- If validation fails (like missing name or invalid email), Spring throws an exception called MethodArgumentNotValidException.

- Without handling it → you'll get a long ugly stack trace 😩

- With @RestControllerAdvice → you can send a neat response ✅

🌍 Create Global Exception Handler :

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;

import java.util.HashMap;
import java.util.Map;

@RestControllerAdvice
public class GlobalExceptionHandler {

    // ✅ Handle validation errors
    @ExceptionHandler(MethodArgumentNotValidException.class)
     public ResponseEntity<Map<String, String>>
handleValidationErrors(MethodArgumentNotValidException ex) {

        Map<String, String> errors = new HashMap<>();

        ex .getBindingResult().getFieldErrors().forEach(error -> {
            errors.put(error.getField(), error.getDefaultMessage());
        });

        return new ResponseEntity<>(errors, HttpStatus.BAD_REQUEST);
    }

    // ✅ Handle other exceptions (like NullPointerException)
    @ExceptionHandler(Exception.class)
    public ResponseEntity<Map<String, String>> handleGlobalExceptions(Exception ex) {

        Map<String, String> error = new HashMap<>();
        error.put("error", ex.getMessage());

        return new ResponseEntity<>(error, HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

#. Example Output (for invalid input)

If the client sends:

```
{
  "name": "",
  "email": "ajaygmail.com",
  "age": 15
}
```

==> Then your API responds with a clean error:

```
{
 "name": "Name is required",
 "email": "Invalid email format",
 "age": "Age must be at least 18"
}
```

==> ✅ Much cleaner and user-friendly than a raw stack trace!

#. Example: Handling Custom Exception :

==> You can also define your own exception 👇

#. Custom Exception:

```
        public class ResourceNotFoundException extends RuntimeException {
        public ResourceNotFoundException(String message) {
         super(message);
       }
   }
```

==> Throw it in Service:

```
      if (user == null) {
          throw new ResourceNotFoundException("User not found with id: " + id);
    }
```

#. Handle it in Global Handler:

```
        @ExceptionHandler(ResourceNotFoundException.class)

         public ResponseEntity<Map<String, String>>
handleResourceNotFound(ResourceNotFoundException ex) {

        Map<String, String> error = new HashMap<>();

        error.put("error", ex.getMessage());

      return new ResponseEntity<>(error, HttpStatus.NOT_FOUND);
}
```

Output:

```
  {
 "error": "User not found with id: 10"
 }
```

Q: What Happens Internally :

When an exception occurs:

1. Spring looks for a matching @ExceptionHandler in the same controller.
2. If not found → it checks the global @RestControllerAdvice.
3. If still not found → default Spring Boot error response is returned.

Q: What's differences Difference Between @ControllerAdvice and @RestControllerAdvice ?

@ControllerAdvice	@RestControllerAdvice
Used in MVC apps returning views	Used in REST APIs returning JSON
Combines @ControllerAdvice + @ResponseBody manually	Automatically includes @ResponseBody
Returns ModelAndView objects	Returns JSON responses

==> In REST APIs, always use @RestControllerAdvice.

#. @ExceptionHandler :

- @ExceptionHandler is an annotation in Spring Boot used to handle specific exceptions in a controller or globally using @RestControllerAdvice.

==> In simple words:

- It catches a specific exception type and defines what response to return when that exception happens.

Package:

import org.springframework.web.bind.annotation.ExceptionHandler;

#. You Can Handle Multiple Exceptions Together

==> If you want one handler for several exceptions 👇

```
@ExceptionHandler({NullPointerException.class, IllegalArgumentException.class})

public ResponseEntity<Map<String, String>> handleCommonExceptions(Exception ex) {

    Map<String, String> error = new HashMap<>();

    error.put("error", ex.getMessage());

    return new ResponseEntity<>(error, HttpStatus.BAD_REQUEST);
}
```

Summary :

- @ExceptionHandler catches specific exceptions and returns custom responses.
- It can be placed inside a controller (local) or inside a global advice class.
- Commonly paired with @RestControllerAdvice for centralized global exception handling.
- You can return custom messages, HTTP codes, and structured JSON.

Q:

What is Spring Security?

- Spring Security is a powerful authentication and authorization framework used to secure Java and Spring Boot applications.

- It provides protection against unauthorized access, and common attacks like CSRF, XSS, and session fixation.

Authentication :

- 👉 Who are you?
- It verifies the user's identity (like username & password).

Example:

Login form → username/password → system verifies credentials.

#. Spring Security uses:

- UserDetailsService to load user info from DB

- AuthenticationManager to verify credentials

Authorization :

- 👉 What are you allowed to do?
- After authentication, it decides what resources the user can access.

For example:

- Admin → can access everything
- User → limited access

==> It uses roles like: ROLE_ADMIN, ROLE_USER

Q. What are the main features of Spring Security?

1. Authentication (who you are)
2. Authorization (what you can do)
3. Password encryption
4. Role-based access control
5. Protection from CSRF, XSS, Session Hijacking
6. Integration with OAuth2, JWT, LDAP, etc.

#. Security Filters :

- Spring Security works as a chain of filters that intercept every HTTP request.

#. Common filters:

- UsernamePasswordAuthenticationFilter → handles login

- BasicAuthenticationFilter → for basic auth

- SecurityContextPersistenceFilter → stores security context

Q: What is SecurityContextPersistenceFilter?

- SecurityContextPersistenceFilter is one of the first filters in the Spring Security filter chain.
- It is responsible for loading and storing the SecurityContext for each user request.

Q: What is a SecurityContext?

==> A SecurityContext holds the security information of the currently authenticated user — like:

- Username
- Roles (authorities)
- Authentication status

==> This context is stored in the SecurityContextHolder.

Q: What is SecurityContextHolder?

- It stores information about the currently authenticated user.
- You can access it anywhere in your app using :

  SecurityContextHolder.getContext().getAuthentication();

Q: How it works (Step-by-Step):

1. Incoming request :

==> When a new HTTP request comes in:

- SecurityContextPersistenceFilter checks if there's already an existing SecurityContext (like from session or token).
- If found, it loads it into the SecurityContextHolder.
- If not, it creates a new empty context.

2. Request Processing :

==>Then the request passes through other filters:

- UsernamePasswordAuthenticationFilter → authenticates credentials.
- BasicAuthenticationFilter → handles HTTP Basic auth, etc.

Once the user is authenticated, the authentication details are stored inside the SecurityContext.

3. After the request completes :

==> At the end of request processing:

- SecurityContextPersistenceFilter clears or saves the context back to the session (if session-based).
- This ensures that on the next request, the user doesn't need to log in again (if session exists).

⚠️ **Important Interview Tip:**

- SecurityContextPersistenceFilter runs at the beginning and end of every request.
- It ensures that the SecurityContextHolder always has the correct authentication for the current thread.
- Without this filter, Spring Security wouldn't know who is making the request.

Q What is UserDetailsService?

- It's an interface used to load user details (username, password, roles) from the database or any custom source.
- Custom class implements this interface to provide user authentication logic.

Q: What is PasswordEncoder and why is it used?

- Used to encrypt passwords before saving to DB and during login comparison.

Example:

```
new BCryptPasswordEncoder()
```

==> BCrypt is the most commonly used secure algorithm.

==> ✅ Secure way — never store plain-text passwords.

Q: What is CSRF in Spring Security?

- CSRF (Cross-Site Request Forgery) is an attack where a malicious site tricks a logged-in user to perform unintended actions.
- Spring Security provides built-in CSRF protection, mainly for web apps (not REST APIs).
- Spring Security automatically protects from Cross-Site Request Forgery attacks.
- "In Spring Security, CSRF protection is enabled by default.
- To disable it, we use **.csrf().disable()** in HttpSecurity configuration.
- It is usually disabled for stateless REST APIs where authentication is handled by tokens."
- ⚠️ Do not disable CSRF for traditional web applications unless you have a good reason.

#. Common Authentication Methods :

- 🔹 Form-based login — default login page
- 🔹 Basic Auth — used in REST APIs
- 🔹 JWT (JSON Web Token) — for token-based authentication

Q: How Spring Security Works (Flow) ?

User sends login request (username/password)

UsernamePasswordAuthenticationFilter captures login credentials.

AuthenticationManager validates credentials (using UserDetailsService).

If valid → creates an Authentication object stored in SecurityContextHolder.

Authorization filters check user roles before accessing resources.

Q. How do you secure REST APIs using Spring Security?

- Use Basic Auth or JWT Token Authentication
- Annotate endpoints with role-based access:
  `@PreAuthorize("hasRole('ADMIN')")`
- Configure a SecurityFilterChain or WebSecurityConfigurerAdapter (Spring Boot 2.x)

Q How to customize Spring Security login?

- You can create a custom login form or disable the default one using:
  `http.formLogin().loginPage("/login");`

#. Bonus Tip (Interview Insight):

- In REST APIs, JWT Authentication is preferred.
- In web apps, form-based login is common.
- Always encrypt passwords using BCryptPasswordEncoder.

Q: What is SecurityFilterchain in spring security ?

- "SecurityFilterChain is the core component in Spring Security that holds the chain of filters to process HTTP requests.
- It allows us to define authentication, authorization, CSRF, CORS, and other security rules for our application."

💡 Pro Interview Tip:

- From Spring Security 5.7+, SecurityFilterChain replaced WebSecurityConfigurerAdapter.
- So if asked in interview — mention that this is the modern approach for configuring security.

Q:
"How do you provide authentication and authorization in Spring Security?"

#. First provide Authentication in Spring Security :

- **There are 3 main ways:**

(a) In-Memory Authentication (for demo/testing)

- **You hardcode users and roles directly in your configuration class:**

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public UserDetailsService userDetailsService() {

        UserDetails admin = User.withUsername("ajay")
                .password(passwordEncoder().encode("1234"))
                .roles("ADMIN")
                .build();

        UserDetails user = User.withUsername("kumar")
                .password(passwordEncoder().encode("1234"))
                .roles("USER")
                .build();

        return new InMemoryUserDetailsManager(admin, user);
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

Explanation:

- The UserDetailsService provides user info to Spring Security.
- The PasswordEncoder encrypts passwords (BCrypt is standard).
- Spring Security automatically authenticates users when they log in.

(b) Database (Custom) Authentication

- For real projects, users are stored in a database.
- You use your own User entity and UserDetailsService implementation.

```
@Service
public class CustomUserDetailsService implements UserDetailsService {

   @Autowired
   private UserRepository userRepository;

   @Override
   public UserDetails loadUserByUsername(String username)
         throws UsernameNotFoundException {

      UserEntity user = userRepository.findByUsername(username)
            .orElseThrow(() -> new UsernameNotFoundException("User not found"));

             return new org.springframework.security.core.userdetails.User(
                   user.getUsername(),
                   user.getPassword(),
                  getAuthorities(user)
          );
   }

   private Collection<? extends GrantedAuthority> getAuthorities(UserEntity user) {

       return user.getRoles().stream()
             .map(role -> new SimpleGrantedAuthority("ROLE_" + role))
             .collect(Collectors.toList());
   }
}
```

#. Explanation:

- Fetch user details from DB
- Return UserDetails object containing username, password, and roles
- Spring Security uses this for authentication.

(c) Token-Based Authentication (JWT) :

- Used in REST APIs.
- Instead of sessions, you send a JWT token in each request header.
- Spring Security validates it before allowing access.

Second Authorization in Spring Security :

- After authentication, Spring Security applies access control rules.
- You can provide authorization in two main ways:

// Q: How do you configure role-based authorization in spring security .

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
   http
      .authorizeHttpRequests()
      .requestMatchers("/admin/**").hasRole("ADMIN")
      .requestMatchers("/user/**").hasAnyRole("USER", "ADMIN")
      .anyRequest().authenticated()
      .and()
      .formLogin()
      .and()
      .httpBasic();
   return http.build();
}
```

#. Explanation:

- /admin/** → only ADMIN can access
- /user/** → USER or ADMIN can access
- .anyRequest().authenticated() → all other endpoints need login

(b) Method-Level Authorization :

- Use annotations on controller or service methods:

```
@PreAuthorize("hasRole('ADMIN')")
@GetMapping("/admin")
public String adminOnly() {
   return "Welcome Admin!";
}
```

✅ **Enable it in your configuration:**

@EnableMethodSecurity(prePostEnabled = true)

Summary (Interview Answer):

In Spring Security,

- Authentication is provided by UserDetailsService (which loads user data from DB, memory, or external source).
- Authorization is provided via role-based access control using HttpSecurity configurations or annotations like @PreAuthorize.
- Passwords are encrypted using a PasswordEncoder (commonly BCrypt).
- Once authenticated, user details are stored in SecurityContextHolder and used for subsequent requests.

Q:

What is JWT?

- JWT (JSON Web Token) is a compact, digitally signed token used to verify the identity of a user and securely transmit information between client and server.
- It's divided into three parts:

  xxxxx . yyyyy . zzzzz

  Header . Payload . Signature

#. Example:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiJhamF5QGdtYWlsLmNvbSIsInJvbGUiOiJBRE1JTiIsImlhdCI6MTY5MzExNTI1OCwiZXhwIjoxNjkzMTIzMDU4fQ.
SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

Q: How JWT Works in Spring Security

Here's the flow 👇

🧭 Authentication Flow:

1. User logs in with username/password → POST /api/auth/login

2. Spring authenticates credentials.

3. If valid → generate JWT token.

4. Return JWT token to the client.

5. Client includes this JWT in Authorization header for subsequent requests:

Authorization: Bearer <token>

6. The JWT is validated in each request by a JWT Filter before accessing secured endpoints.

Q: How to configure JWT-Based Authentication & Authorization in Spring Security ?

Steps1 --- to Configure JWT in Spring Security :

```
<dependency>
   <groupId>io.jsonwebtoken</groupId>
   <artifactId>jjwt</artifactId>
   <version>0.9.1</version>
</dependency>
```

Step 2 — Create JWT Utility Class

- Handles token creation and validation.

```
@Component
public class JwtUtil {
   private static final String SECRET_KEY = "my_secret_key";

   public String generateToken(UserDetails userDetails) {
      return Jwts.builder()
            .setSubject(userDetails.getUsername())
            .claim("roles", userDetails.getAuthorities())
            .setIssuedAt(new Date(System.currentTimeMillis()))
            .setExpiration(new Date(System.currentTimeMillis() + 1000 * 60 * 60)) // 1 hour
            .signWith(Keys.hmacShaKeyFor(SECRET_KEY.getBytes()), SignatureAlgorithm.HS256)
            .compact();
   }
   public String extractUsername(String token) {
      return Jwts.parserBuilder()
            .setSigningKey(SECRET_KEY.getBytes())
            .build()
            .parseClaimsJws(token)
            .getBody()
            .getSubject();
   }
   public boolean isTokenValid(String token, UserDetails userDetails) {
      String username = extractUsername(token);
      return username.equals(userDetails.getUsername()) && !isTokenExpired(token);
   }
   private boolean isTokenExpired(String token) {
      Date expiration = Jwts.parserBuilder()
            .setSigningKey(SECRET_KEY.getBytes())
            .build()
            .parseClaimsJws(token)
            .getBody()
            .getExpiration();
      return expiration.before(new Date());
   }
}
```

Step 3: Create JwtAuthenticationFilter.java

- Used to intercept every request and validate JWT token.

Program Example :

```
@Component
public class JwtAuthenticationFilter extends OncePerRequestFilter {

    @Autowired
    private JwtUtil jwtUtil;
    @Autowired
    private UserDetailsService userDetailsService;

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
                                    FilterChain filterChain) throws ServletException, IOException {

        String authHeader = request.getHeader("Authorization");
        String token = null;
        String username = null;

        if (authHeader != null && authHeader.startsWith("Bearer ")) {
            token = authHeader.substring(7);
            username = jwtUtil.extractUsername(token);
        }

        if (username != null && SecurityContextHolder.getContext().getAuthentication() == null) {
            UserDetails userDetails = userDetailsService.loadUserByUsername(username);

            if (jwtUtil.isTokenValid(token, userDetails)) {
                UsernamePasswordAuthenticationToken authToken =
                        new UsernamePasswordAuthenticationToken(
                                userDetails, null, userDetails.getAuthorities());

                authToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));

                SecurityContextHolder.getContext().setAuthentication(authToken);
            }
        }
        filterChain.doFilter(request, response);
    }
}
```

Step 4: Security Configuration (SecurityConfig.java) :

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

   @Autowired
   private JwtAuthenticationFilter jwtAuthFilter;

   @Bean
   public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
      http
         .csrf(csrf -> csrf.disable())

         .authorizeHttpRequests(auth -> auth

            .requestMatchers("/api/auth/**")

            .permitAll()

            .anyRequest().authenticated()
         )

         .sessionManagement(sess ->
sess.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
         .addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class);

      return http.build();
   }

   @Bean
   public PasswordEncoder passwordEncoder() {

      return new BCryptPasswordEncoder();

   }
}
```

Step 5: Authentication Controller

```
@RestController
@RequestMapping("/api/auth")
public class AuthController {

   @Autowired
   private AuthenticationManager authenticationManager;

   @Autowired
   private JwtUtil jwtUtil;

   @Autowired
   private UserDetailsService userDetailsService;

   @PostMapping("/login")
   public ResponseEntity<String> login(@RequestBody AuthRequest request) {

  authenticationManager.authenticate(
     new UsernamePasswordAuthenticationToken(request.getUsername(), request.getPassword())
      );

      UserDetails userDetails = userDetailsService.loadUserByUsername(request.getUsername());

      String token = jwtUtil.generateToken(userDetails);

      return ResponseEntity.ok(token);
   }
}
```

Step 5: DTO Class for Login

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class AuthRequest {
   private String username;
   private String password;
}
```

#. Request Flow Example

1. POST /api/auth/login → returns JWT.

2. GET /api/users → send header:
   Authorization: Bearer <token>

3. If token is valid → access granted; else → 401 Unauthorized.

🔒 6. Advantages of JWT-Based Security :

✅ Stateless → no session stored on server

✅ Faster Authentication

✅ Works across distributed systems (Microservices)

✅ Scalable and widely supported

Question:

Why do we use OncePerRequestFilter for JWT validation?

- Because it ensures the filter executes exactly once per request, which is perfect for stateless authentication.

#. Different case of JWT :

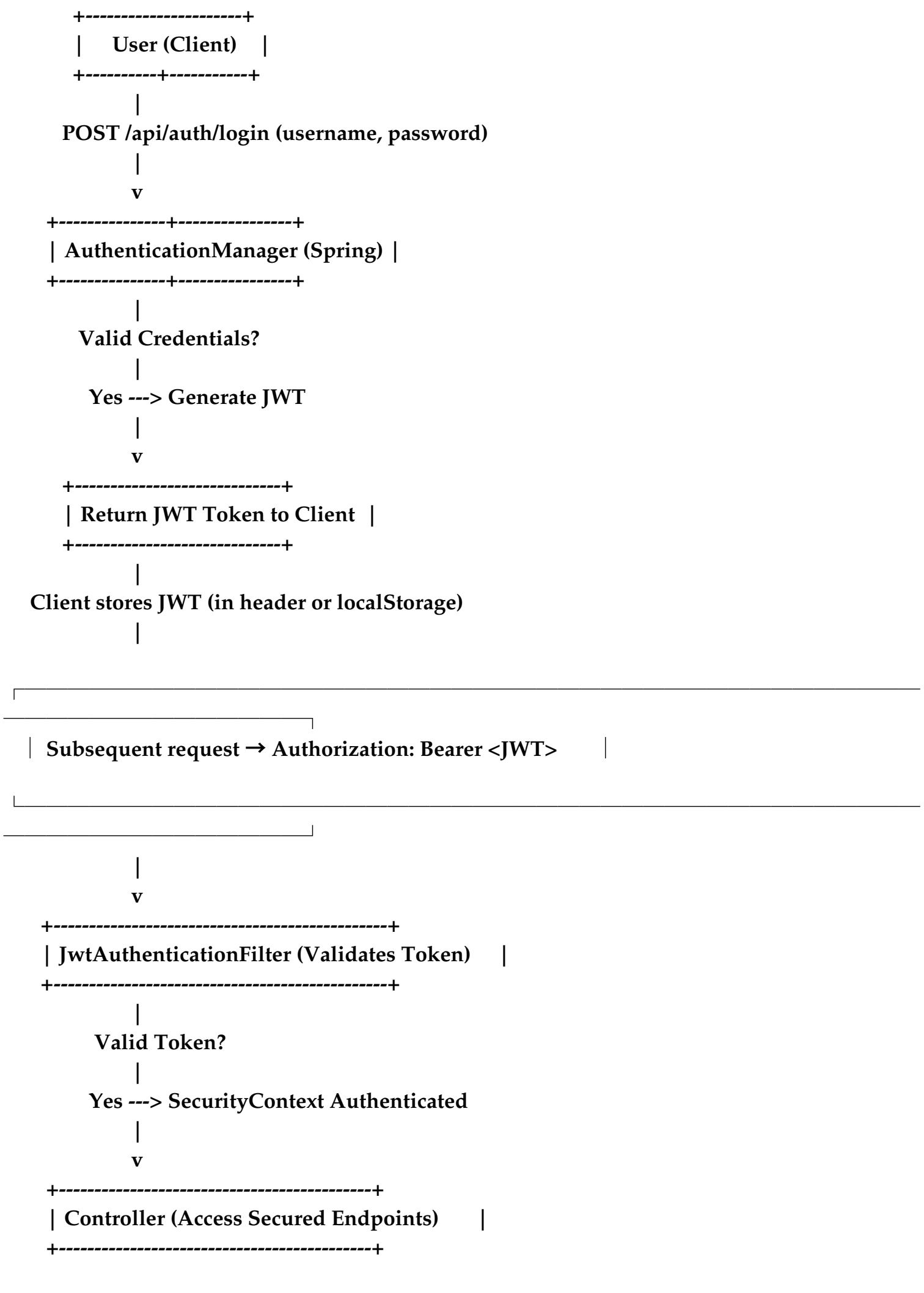
- **Let's now go step-by-step through a complete Spring Boot JWT Authentication & Authorization Project Structure, including folder layout, code flow, and how all components interact in real-world projects — exactly the way interviewers expect you to explain it.**

Project Folder Structure (JWT Auth Example) :

- **Here's the standard folder layout for a Spring Boot JWT-based Authentication & Authorization project:**

```
springboot-jwt-auth/
│
├── src/main/java/com/example/jwtsecurity/
│   ├── config/
│   │   └── SecurityConfig.java
│   │
│   ├── controller/
│   │   └── AuthController.java
│   │
│   ├── entity/
│   │   └── UserEntity.java
│   │
│   ├── repository/
│   │   └── UserRepository.java
│   │
│   ├── service/
│   │   ├── CustomUserDetailsService.java
│   │   └── JwtService.java
│   │
│   ├── filter/
│   │   └── JwtAuthenticationFilter.java
│   │
│   ├── model/
│   │   └── AuthRequest.java
│   │
│   ├── controller/
│   │   └── UserController.java
│   │
│   └── JwtSecurityApplication.java
│
└── pom.xml
```

Flow Diagram — JWT Authentication & Authorization



File-by-File Breakdown

🏁 JwtSecurityApplication.java

==> Main entry point.

```
 @SpringBootApplication
public class JwtSecurityApplication {

   public static void main(String[] args) {

      SpringApplication.run(JwtSecurityApplication.class, args);
   }
}
```

👤 UserEntity.java

```
@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
@Table(name = "users")
public class UserEntity {

   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;

   private String username;

   private String password;

   private String role;

}
```

💾 UserRepository.java

```
      public interface UserRepository extends JpaRepository<UserEntity, Long> {

            Optional<UserEntity> findByUsername(String username);
}
```

🧠 CustomUserDetailsService.java

Implements UserDetailsService to load user data from DB.

```
@Service
public class CustomUserDetailsService implements UserDetailsService {

 @Autowired
 private UserRepository repository;

 @Override
 public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {

 UserEntity user = repository.findByUsername(username)
        .orElseThrow(() -> new UsernameNotFoundException("User not found: " + username));

      return new org.springframework.security.core.userdetails.User(
         user.getUsername(),

         user.getPassword(),

        Collections.singleton(new SimpleGrantedAuthority(user.getRole()))
   );
 }
}
```

🔐 **JwtService.java**

Generates and validates JWT.

```
@Service
public class JwtService {

   private static final String SECRET_KEY = "my_secret_key_123456789";

   public String generateToken(UserDetails userDetails) {
      return Jwts.builder()
            .setSubject(userDetails.getUsername())
            .claim("roles", userDetails.getAuthorities())
            .setIssuedAt(new Date(System.currentTimeMillis()))
            .setExpiration(new Date(System.currentTimeMillis() + 1000 * 60 * 60)) // 1 hr
            .signWith(Keys.hmacShaKeyFor(SECRET_KEY.getBytes()),
SignatureAlgorithm.HS256)
            .compact();
   }

   public String extractUsername(String token) {
      return Jwts.parserBuilder()
            .setSigningKey(SECRET_KEY.getBytes())
            .build()
            .parseClaimsJws(token)
            .getBody()
            .getSubject();
   }

   public boolean isTokenValid(String token, UserDetails userDetails) {
      return userDetails.getUsername().equals(extractUsername(token))
            && !isTokenExpired(token);
   }

   private boolean isTokenExpired(String token) {
      return Jwts.parserBuilder()
            .setSigningKey(SECRET_KEY.getBytes())
            .build()
            .parseClaimsJws(token)
            .getBody()
            .getExpiration()
            .before(new Date());
   }
}
```

🧱 **JwtAuthenticationFilter.java**

Intercepts requests and validates JWT.

```
 @Component
public class JwtAuthenticationFilter extends OncePerRequestFilter {

   @Autowired
   private JwtService jwtService;
   @Autowired
   private CustomUserDetailsService userDetailsService;

   @Override
   protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
                                   FilterChain filterChain)
         throws ServletException, IOException {

      String authHeader = request.getHeader("Authorization");
      String token = null;
      String username = null;

      if (authHeader != null && authHeader.startsWith("Bearer ")) {
         token = authHeader.substring(7);
         username = jwtService.extractUsername(token);
      }

      if (username != null && SecurityContextHolder.getContext().getAuthentication() == null) {
         UserDetails userDetails = userDetailsService.loadUserByUsername(username);
         if (jwtService.isTokenValid(token, userDetails)) {
            UsernamePasswordAuthenticationToken authToken =
                  new UsernamePasswordAuthenticationToken(userDetails, null,
userDetails.getAuthorities());
            authToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));

            SecurityContextHolder.getContext().setAuthentication(authToken);
         }
      }
      filterChain.doFilter(request, response);
   }
}
```

⚙ SecurityConfig.java

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

   @Autowired
   private JwtAuthenticationFilter jwtAuthFilter;

   @Bean
   public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
      http
         .csrf(csrf -> csrf.disable())

         .authorizeHttpRequests(auth -> auth

            .requestMatchers("/api/auth/**").permitAll()

            .anyRequest().authenticated())

         .sessionManagement(sess ->
 sess.sessionCreationPolicy(SessionCreationPolicy.STATELESS))

         .addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class);

      return http.build();
   }

   @Bean
   public AuthenticationManager authenticationManager(AuthenticationConfiguration config)
         throws Exception {
            return config.getAuthenticationManager();
   }

   @Bean
   public PasswordEncoder passwordEncoder() {

      return new BCryptPasswordEncoder();

   }
}
```

AuthRequest.java

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class AuthRequest {
   private String username;
   private String password;
}
```

AuthController.java

- Handles login and token generation.

```
@RestController
@RequestMapping("/api/auth")
public class AuthController {

   @Autowired
   private AuthenticationManager authenticationManager;
   @Autowired
   private CustomUserDetailsService userDetailsService;
   @Autowired
   private JwtService jwtService;

   @PostMapping("/login")
   public ResponseEntity<String> login(@RequestBody AuthRequest request) {
      authenticationManager.authenticate(
            new UsernamePasswordAuthenticationToken(request.getUsername(),
request.getPassword()));

      UserDetails userDetails = userDetailsService.loadUserByUsername(request.getUsername());
      String token = jwtService.generateToken(userDetails);

      return ResponseEntity.ok(token);
   }
}
```

UserController.java

- Protected endpoint — requires valid JWT.

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

   @GetMapping("/welcome")
   public ResponseEntity<String> welcome() {
      return ResponseEntity.ok("Welcome, Authorized User!");
   }
}
```

Example API Calls

🔹 **Step 1 — Login to get token:**

POST /api/auth/login

```
{
  "username": "ajay",
  "password": "password123"
}
```

Response:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...

🔹 **Step 2 — Use token for access:**

GET /api/users/welcome

Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...

Response:

Welcome, Authorized User!

🧠 **Key Interview Points to Remember**

Concept	Explanation
Stateless Auth	JWT removes need for server sessions
Filter Chain	JWT filter executes before username-password filter
OncePerRequestFilter	Runs exactly once per request
SessionCreationPolicy.STATELESS	Ensures no session stored in memory
@EnableWebSecurity	Enables Spring Security Configuration
Token Expiry	Always include exp claim for security

Q: How will you add JWT Refresh Token Implementation next (so users can renew expired tokens without logging in again)?

==> That's often the next-level interview question after JWT basics ✅

- Let's go step-by-step, so you'll understand why we use refresh tokens, how they work, and how to implement them in a clean Spring Boot project.

What is a Refresh Token?

- A Refresh Token is a long-lived token used to generate a new Access Token (JWT) when the original token expires.

🧠 Problem:

- Access Tokens usually expire in 15–30 minutes for security.
- If a token expires, the user would need to log in again — inconvenient.

✅ Solution:

We issue two tokens during login:

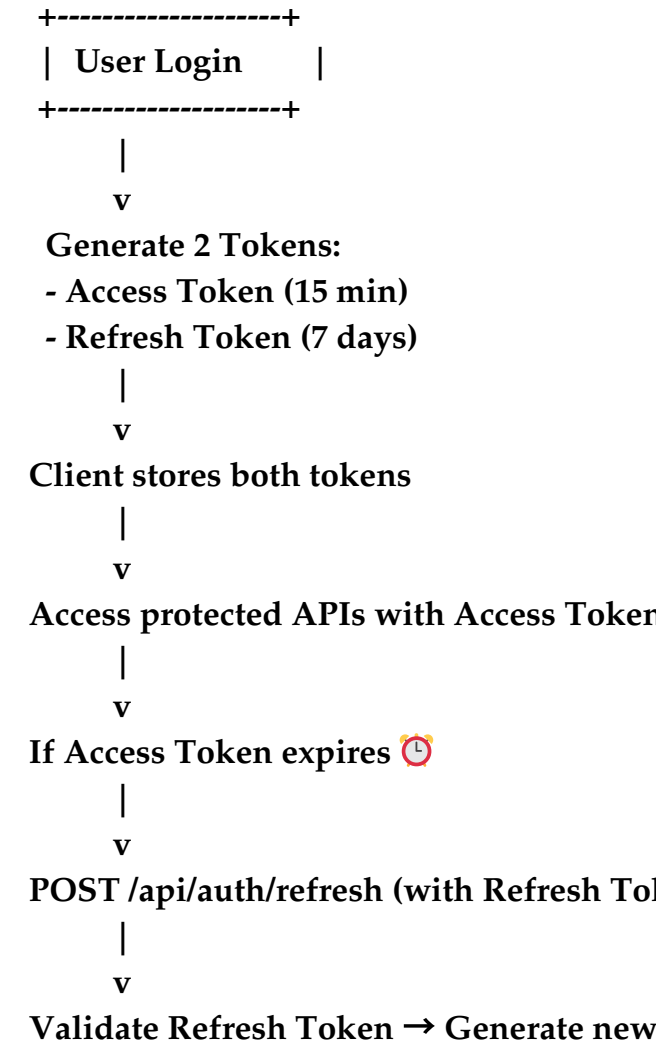
1. Access Token — short-lived (e.g., 15 min)
2. Refresh Token — long-lived (e.g., 7 days)

==> When Access Token expires, the user calls:

POST /api/auth/refresh

==> with the Refresh Token, and receives a new Access Token.

🔁 How Refresh Token Flow Works ?



⚙️ Code Implementation

- We'll build on top of your existing JWT setup.

✅ Step 1: Update JwtService.java

- Add refresh token generation & validation.

```
@Service
public class JwtService {
    private static final String SECRET_KEY = "my_secret_key_123456789";
    // 🔹 Generate Access Token (short-lived)
    public String generateAccessToken(UserDetails userDetails) {
        return Jwts.builder()
                .setSubject(userDetails.getUsername())
                .claim("roles", userDetails.getAuthorities())
                .setIssuedAt(new Date(System.currentTimeMillis()))
                .setExpiration(new Date(System.currentTimeMillis() + 1000 * 60 * 15)) // 15 mins
                .signWith(Keys.hmacShaKeyFor(SECRET_KEY.getBytes()), SignatureAlgorithm.HS256)
                .compact();
    }
    // 🔹 Generate Refresh Token (long-lived)
    public String generateRefreshToken(UserDetails userDetails) {
        return Jwts.builder()
                .setSubject(userDetails.getUsername())
                .setIssuedAt(new Date(System.currentTimeMillis()))
                .setExpiration(new Date(System.currentTimeMillis() + 1000L * 60 * 60 * 24 * 7)) // 7 days
                .signWith(Keys.hmacShaKeyFor(SECRET_KEY.getBytes()), SignatureAlgorithm.HS256)
                .compact();
    }
    // 🔹 Extract Username
    public String extractUsername(String token) {
        return Jwts.parserBuilder()
                .setSigningKey(SECRET_KEY.getBytes())
                .build()
                .parseClaimsJws(token)
                .getBody()
                .getSubject();
    }
    // 🔹 Check Expiration
    private boolean isTokenExpired(String token) {
        Date expiration = Jwts.parserBuilder()
                .setSigningKey(SECRET_KEY.getBytes())
                .build()
                .parseClaimsJws(token)
                .getBody()
                .getExpiration();
        return expiration.before(new Date());
    }
    // 🔹 Validate Token
    public boolean isTokenValid(String token, UserDetails userDetails) {
        return extractUsername(token).equals(userDetails.getUsername()) && !isTokenExpired(token);
    }
}
```

✅ Step 2: Create DTO for Refresh Request & Response
RefreshTokenRequest.java

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class RefreshTokenRequest {
    private String refreshToken;
}
```

AuthResponse.java

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class AuthResponse {
    private String accessToken;
    private String refreshToken;
}
```

✅ Step 3: Update AuthController.java
Add /refresh endpoint and return both tokens during login.

```
@RestController
@RequestMapping("/api/auth")
public class AuthController {

    @Autowired
    private AuthenticationManager authenticationManager;
    @Autowired
    private CustomUserDetailsService userDetailsService;
    @Autowired
    private JwtService jwtService;

    @PostMapping("/login")
    public ResponseEntity<AuthResponse> login(@RequestBody AuthRequest request) {
        authenticationManager.authenticate(
                new UsernamePasswordAuthenticationToken(request.getUsername(),
request.getPassword())
        );

        UserDetails userDetails = userDetailsService.loadUserByUsername(request.getUsername());

        String accessToken = jwtService.generateAccessToken(userDetails);
        String refreshToken = jwtService.generateRefreshToken(userDetails);

        return ResponseEntity.ok(new AuthResponse(accessToken, refreshToken));
    }
```

```
 @PostMapping("/refresh")
 public ResponseEntity<AuthResponse> refreshToken(@RequestBody RefreshTokenRequest
request) {
 String refreshToken = request.getRefreshToken();
 String username = jwtService.extractUsername(refreshToken);

 UserDetails userDetails = userDetailsService.loadUserByUsername(username);

 if (jwtService.isTokenValid(refreshToken, userDetails)) {
 String newAccessToken = jwtService.generateAccessToken(userDetails);
 return ResponseEntity.ok(new AuthResponse(newAccessToken, refreshToken));
 } else {
 return ResponseEntity.status(HttpStatus.UNAUTHORIZED).build();
 }
 }
}
```

Example API Usage

🔹 Login (Generate both tokens)

POST /api/auth/login

```
    {
        "username": "ajay",
         "password": "password123"
   }
```

✅ Response:

```
    {
       "accessToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
      "refreshToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."
 }
```

🔹 Use Access Token

GET /api/users/welcome

Authorization: Bearer <access_token>

If expired → get 401 Unauthorized.

🔹 Refresh Token Flow

POST /api/auth/refresh

```
    {
      "refreshToken": "<refresh_token>"
   }
```

✅ Response:

```
{
 "accessToken": "new_access_token_here",
 "refreshToken": "same_refresh_token"
}
```

🧠 Important Interview Tips

Question	Answer
❓ **Why two tokens?**	Access token for quick auth, refresh token for long-term session
❓ **Why not use a single token?**	Short expiry improves security, refresh token prevents frequent logins
❓ **Where to store tokens on frontend?**	Access token → memory, Refresh token → HttpOnly cookie or secure storage
❓ **Can refresh tokens be revoked?**	Yes, by maintaining a **token blacklist** or storing them in a DB
❓ **What's a safe expiry period?**	Access: 15 min, Refresh: 7 days (configurable)

Q:
What is @AuthenticationPrincipal?

- @AuthenticationPrincipal is an annotation provided by Spring Security.
- It is used to inject the currently authenticated user directly into a controller method.
- This saves you from manually accessing the SecurityContextHolder to get the logged-in user.

Note :

- It helps us to access details of the currently logged-in(authenticated) user easily .

Example 1: Using Default UserDetails

```
import org.springframework.security.core.annotation.AuthenticationPrincipal;
import org.springframework.security.core.userdetails.User;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class UserController {

    @GetMapping("/current-user")
    public String getCurrentUser(@AuthenticationPrincipal User user) {
        return "Hello, " + user.getUsername();
    }
}
```

✅ Here:

- @AuthenticationPrincipal User user automatically injects the authenticated UserDetails object.
- You can directly use **user.getUsername()** or other info.

Example 2: Using Custom User Class

- Suppose you have a custom user class:

```
public class CustomUser {
    private String username;
    private String email;
    // getters and setters
}
```

Controller:

```
@GetMapping("/profile")
public String getProfile(@AuthenticationPrincipal CustomUser user) {
    return "Hello " + user.getUsername() + ", your email is " + user.getEmail();
}
```

✅ Spring injects the logged-in CustomUser object automatically.

Example 3: With JWT or OAuth2

```
@GetMapping("/me")
public String getLoggedInUser(@AuthenticationPrincipal OAuth2User principal) {
   return principal.getAttribute("name");  // get name from OAuth2 token
}
```

Summary

- @AuthenticationPrincipal simplifies access to the logged-in user.
- Works with UserDetails, custom user classes, or OAuth2User.
- **Avoids boilerplate code like:**

```
Authentication auth = SecurityContextHolder.getContext().getAuthentication();

UserDetails user = (UserDetails) auth.getPrincipal();
```

Q:
What is @PostConstruct?

- It helps us to run a method automatically after a bean is created and dependencies are injected when the project start .
- In Spring, it is often used in bean lifecycle management.

#. Example 1: Basic Usage

```
import javax.annotation.PostConstruct;
import org.springframework.stereotype.Component;

@Component
public class SweetShop {

   public SweetShop() {
      System.out.println("Constructor called");
   }

   @PostConstruct
   public void init() {
      System.out.println("PostConstruct method called: initializing SweetShop");
   }
}
```

✅ **Output when Spring starts:**

```
Constructor called
PostConstruct method called: initializing SweetShop
```

Example 2: With Dependency Injection

```
@Component
public class SweetService {

    private final SweetRepository repo;

    public SweetService(SweetRepository repo) {
        this.repo = repo;
    }

    @PostConstruct
    public void setup() {
        System.out.println("SweetService initialized with repo: " + repo);
    }
}
```

✅ **Here:**

- Spring injects SweetRepository first
- Then setup() runs automatically

🔹 Important Notes

- Only one method per bean should be annotated with @PostConstruct.
- Method cannot take parameters and must be void.
- Runs after constructor and dependency injection, but before the bean is ready for use.
- In Spring Boot, often used to load default data, initialize caches, or set up resources.

Q:

What is ModelMapper?

- ModelMapper is not a Spring annotation — it's a Java library used to automatically map objects (usually between Entity, DTO, or VO classes).
- It helps to reduce boilerplate code for copying data from one object to another.

Note :

- It hepls us to convert entity object to DTO , which is used for sending data to the frontend .

Example Problem (Without ModelMapper)

- **Suppose you have:**

```
public class UserEntity {

   private Long id;
   private String username;
   private String email;

}
```

==> and a DTO:

```
    public class UserDTO {
        private String username;
        private String email;
   }
```

==> Normally, you would do this manually:

```
       UserDTO dto = new UserDTO();
       dto.setUsername(entity.getUsername());
       dto.setEmail(entity.getEmail());
```

==> ❌ Boring and repetitive for big projects.

✅ Using ModelMapper :

- You can use the ModelMapper library to do this automatically.

Step 1: Add Dependency (Maven)

```
<dependency>
   <groupId>org.modelmapper</groupId>
   <artifactId>modelmapper</artifactId>
   <version>3.2.0</version>
</dependency>
```

Step 2: Create a Bean

```
import org.modelmapper.ModelMapper;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class AppConfig {
   @Bean
   public ModelMapper modelMapper() {
      return new ModelMapper();
   }
}
```

Step 3: Use in Service Layer

```
import org.modelmapper.ModelMapper;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class UserService {
    @Autowired
    private ModelMapper modelMapper;

    public UserDTO convertToDto(UserEntity userEntity) {
        return modelMapper.map(userEntity, UserDTO.class);
    }
}
```

✅ Here:

- modelMapper.map(source, destinationType)
- automatically maps all matching field names and types.

Q:

What is @Value in Spring?

- @Value is a Spring annotation used for injecting values into fields, method parameters, or constructor arguments.

- You can use it to inject values from:
  - application.properties / application.yml
  - environment variables
  - system properties

Example 1: Inject value from application.properties

```
app.name=SweetShop
app.version=1.0
```

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class AppInfo {

    @Value("${app.name}")
    private String appName;

    @Value("${app.version}")
    private String version;

    public void printInfo() {
        System.out.println(appName + " - Version: " + version);
    }
}
```

✅ **Output:**

SweetShop - Version: 1.0

Example 2: Inject Default Value

- **If the property is not found, you can provide a default:**

```
@Value("${app.description:Default SweetShop App}")
private String description;
```

- ✅ If app.description is missing, it will use "Default SweetShop App".

Example 3: Inject from Environment Variables

```
@Value("${JAVA_HOME}")
private String javaHome;
```

- ✅ This will inject your system's JAVA_HOME value.

Example 5: Inject Value into Constructor

```
@Component
public class ServerConfig {

   private final String url;

   public ServerConfig(@Value("${server.url}") String url) {
      this.url = url;
   }
}
```

Q:

What is @ModelAttribute?

- @ModelAttribute is a Spring MVC annotation used to bind incomming form or request data (HTTP request parameters) to a Java object (model), and also to expose model data to a view (like Thymeleaf, JSP, etc.).

- It's most often used in controllers — for binding or sharing data between the view and controller.

🔹 Use Case 1: Bind Form Data to Object

Example: A user submits a form with name and email.

HTML Form:

```
<form action="/register" method="post">
   Name: <input type="text" name="name">
   Email: <input type="text" name="email">
   <input type="submit" value="Register">
</form>
```

Java Class:

```
public class User {
   private String name;
   private String email;

   // getters and setters
}
```

Controller:

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PostMapping;

@Controller
public class UserController {

   @PostMapping("/register")
   public String registerUser(@ModelAttribute User user) {
      System.out.println(user.getName() + " - " + user.getEmail());
      return "success";
   }
}
```

✅ Here:

- Spring automatically takes form inputs (name, email)
- and sets them into the User object fields.
- No need for manual parsing with request.getParameter().

Use Case 2: Add Data to Model Before View Loads

- Sometimes, you want to make some data available to every view (like dropdown lists, categories, etc.).

```
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.ui.Model;

@ControllerAdvice
public class GlobalData {

   @ModelAttribute
   public void addCommonData(Model model) {
      model.addAttribute("appName", "SweetShop");
   }
}
```

✅ Here:

- Every page/view rendered will automatically have appName = "SweetShop" available.

Use Case 3: For REST Controllers

- You can still use @ModelAttribute in REST controllers — it binds request parameters to an object even for API calls.

```
@RestController
public class SweetController {

   @PostMapping("/addSweet")

   public String addSweet(@ModelAttribute Sweet sweet) {

      return "Added sweet: " + sweet.getName() + " - " + sweet.getPrice();
   }
}
```

- If you call /addSweet?name=Ladoo&price=50,
- Spring will automatically create a Sweet object with those values.

Q:
What is @Transactional ?

- @Transactional is a Spring annotation used to **manage database transactions automatically.**
- It ensures that a group of database operations (like multiple inserts, updates, or deletes) either all execute successfully or all rollback if any one fails — maintaining data integrity and consistency.

⚙ How it works:

- When a method is annotated with @Transactional, Spring creates a proxy around it.
- The proxy starts a transaction before the method executes.
- If the method executes successfully → Spring commits the transaction.
- If a RuntimeException or unchecked exception occurs → Spring rolls back automatically.

#. Default Behavior:

- Rollback: On RuntimeException and Error.
- No Rollback: On checked exceptions (unless specified using rollbackFor = Exception.class).
- Scope: Usually applied at the service layer.
- Transaction Manager: Spring uses a PlatformTransactionManager (like JpaTransactionManager) internally.

🔹 Common Attributes:

Attribute	Description	Example
propagation	Defines how nested transactions behave	Propagation.REQUIRED (default)
isolation	Controls data visibility in concurrent transactions	Isolation.READ_COMMITTED
rollbackFor	Rollback for specific exceptions	rollbackFor = Exception.class
readOnly	Optimizes performance for read-only queries	readOnly = true

🔹 Key Points to Mention:

- Default scope is method level, but can be applied at class level.
- Ensures ACID properties (Atomicity, Consistency, Isolation, Durability).
- Commonly used with Spring Data JPA, Hibernate, or JDBC.
- Works via Spring AOP (proxy mechanism).
- Should be used in service layer, not in repository.

💻 Project: Spring Boot – Transaction Management Example

🗂️ Folder Structure

```
src/main/java
 ├── com.example.transactiondemo
 │    ├── TransactionDemoApplication.java
 │    ├── entity
 │    │    └── Sweet.java
 │    ├── repository
 │    │    └── SweetRepository.java
 │    ├── service
 │    │    └── SweetService.java
 │    ├── controller
 │    │    └── SweetController.java
```

1. TransactionDemoApplication.java

```
package com.example.transactiondemo;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

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

2. Entity — Sweet.java

```
package com.example.transactiondemo.entity;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;

@Entity
public class Sweet {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private double price;

    public Sweet() {}

    public Sweet(String name, double price) {
        this.name = name;
        this.price = price;
    }

    // getters and setters
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public double getPrice() { return price; }
    public void setPrice(double price) { this.price = price; }
}
```

3. Repository — SweetRepository.java

```
package com.example.transactiondemo.repository;

import com.example.transactiondemo.entity.Sweet;
import org.springframework.data.jpa.repository.JpaRepository;

public interface SweetRepository extends JpaRepository<Sweet, Long> { }
```

4. Service — SweetService.java

```
package com.example.transactiondemo.service;

import com.example.transactiondemo.entity.Sweet;
import com.example.transactiondemo.repository.SweetRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service
public class SweetService {

    @Autowired
    private SweetRepository sweetRepository;

    @Transactional
    public void addTwoSweets() {
        Sweet s1 = new Sweet("Ladoo", 50);
        Sweet s2 = new Sweet("Jalebi", 0);  // invalid (price can't be 0)

        sweetRepository.save(s1);
        System.out.println("Saved Sweet 1");

        // Simulate runtime error
        if (s2.getPrice() <= 0) {
            throw new RuntimeException("Invalid sweet price!");
        }

        sweetRepository.save(s2);
        System.out.println("Saved Sweet 2");
    }
}
```

5. Controller — SweetController.java

```
package com.example.transactiondemo.controller;

import com.example.transactiondemo.service.SweetService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class SweetController {

    @Autowired
    private SweetService sweetService;

    @GetMapping("/save")
    public String saveData() {
        try {
            sweetService.addTwoSweets();
            return "Both sweets saved successfully!";
        } catch (Exception e) {
            return "Transaction rolled back due to: " + e.getMessage();
        }
    }
}
```

6. application.properties

```
spring.datasource.url=jdbc:mysql://localhost:3306/sweetshop
spring.datasource.username=root
spring.datasource.password=root
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

Run & Test

Start Application → Run /save endpoint:

http://localhost:8080/save

✅ Output:

Transaction rolled back due to: Invalid sweet price!

💾 In the database:

- No sweets will be saved because the second insert failed → rollback happened automatically.

Q:
What is a Custom Finder Method?

- In Spring Data JPA, a custom finder method is a method you define in a repository interface (which extends JpaRepository) using a naming convention, and Spring automatically generates the query for you.

- No need to write JPQL or SQL manually.

- Method name defines which fields to search by and what conditions.

◆ Basic Syntax :

List<Entity> findBy<FieldName>(<Type> fieldValue);

- **findBy** is the keyword.

- **<FieldName>** matches the entity's field name.

- Supports logical operators like **And, Or, Between, LessThan, GreaterThan,** etc.

Examples Using Sweet Entity

Assume:

```
@Entity
public class Sweet {
   private String name;
   private double price;
   private String category;
}
```

Find by a single field :

List<Sweet> findByName(String name);

- Finds sweets where name matches exactly.

Find by two fields

List<Sweet> findByCategoryAndPriceLessThan(String category, double price);

- Finds sweets in a category with price less than a given value.

◆ Rules & Notes

1. **Field Names:** Must match entity field names, case-sensitive.
2. **Return Types:** Can be List<T>, Optional<T>, T, or long for counts.
3. **Operators Supported:**
   - And, Or, Between, LessThan, GreaterThan, Like, In, Not, etc.
4. Sorting: Add OrderBy<Field>Asc/Desc

   List<Sweet> findByCategoryOrderByPriceDesc(String category);

#. Spring Data JPA Custom Finder Methods Table

Keyword / Method	Description	Example
findBy	Basic finder by field equality	findByName(String name)
findBy<Field1>And<Field2>	AND condition on multiple fields	findByCategoryAndPrice(String category, double price)
findBy<Field1>Or<Field2>	OR condition	findByCategoryOrPrice(String category, double price)
findBy<Field>Between	Between a range	findByPriceBetween(double min, double max)
findBy<Field>LessThan	Less than condition	findByPriceLessThan(double price)
findBy<Field>LessThanEqual	Less than or equal	findByPriceLessThanEqual(double price)
findBy<Field>GreaterThan	Greater than condition	findByPriceGreaterThan(double price)
findBy<Field>GreaterThanEqual	Greater than or equal	findByPriceGreaterThanEqual(double price)
findBy<Field>Like	SQL LIKE query	findByNameLike("%Ladoo%")
findBy<Field>NotLike	NOT LIKE query	findByNameNotLike("%Ladoo%")
findBy<Field>StartingWith	Starts with	findByNameStartingWith("La")
findBy<Field>EndingWith	Ends with	findByNameEndingWith("di")
findBy<Field>Containing	Contains / LIKE '%keyword%'	findByNameContaining("lad")
findBy<Field>In	Field value in collection	findByCategoryIn(List<String> categories)
findBy<Field>NotIn	Field value not in collection	findByCategoryNotIn(List<String> categories)
findBy<Field>IsNull	Field is null	findByCategoryIsNull()
findBy<Field>IsNotNull	Field is not null	findByCategoryIsNotNull()
findBy<Field>OrderBy<Field>Asc	Order results ascending	findByCategoryOrderByPriceAsc(String category)
findBy<Field>OrderBy<Field>Desc	Order results descending	findByCategoryOrderByPriceDesc(String category)
countBy<Field>	Count rows matching condition	countByCategory(String category)
existsBy<Field>	Check if a record exists	existsByName(String name)
deleteBy<Field>	Delete rows matching condition	deleteByName(String name)
removeBy<Field>	Alias for delete	removeByCategory(String category)

Q:

What is @Query?

- @Query is a Spring Data JPA annotation used to define custom queries (JPQL or native SQL) directly in a repository method.
- It is used when method name conventions (like findBy...) are not sufficient for complex queries.

Syntax

```
@Query("JPQL or SQL query here")

ReturnType methodName(Parameters);
```

- Use JPQL by default (entity names & fields).
- For native SQL, add nativeQuery = true

🔹 Examples

Simple JPQL Query

```
@Query("SELECT s FROM Sweet s WHERE s.category = :category AND s.price < :price")

List<Sweet> findCheapSweets(@Param("category") String category, @Param("price") double price);
```

- **:category and :price are named parameters.**

🔹 Key Notes for Interviews

1. **When to use @Query:**
   - Complex queries (joins, subqueries, aggregation).
   - Queries that cannot be expressed with method naming conventions.

2. JPQL vs Native Query:

Type	Description	Usage
JPQL	Query using **entity names & fields**	Portable & ORM-friendly
Native	Query using **database table/column names**	Database-specific SQL

3. Parameter binding:

- Named (:param)
- Positional (?1, ?2)

4. Return types: List<T>, Optional<T>, T, long, etc.

5. Supports joins & aggregation

Q:

What is Spring Boot Actuator?

- Spring Boot Actuator is a Spring Boot module that provides production-ready features to monitor, manage, and gain insights into your Spring Boot application.
- It exposes health, metrics, environment, and other operational information via HTTP endpoints, JMX, or loggers.
- Commonly used for monitoring, troubleshooting, and performance checks in real-time.

🔹 Key Features

- **Health Checks:** Verify app status (DB, disk, custom checks).
- **Metrics:** JVM stats, CPU, memory, request counts.
- **Environment Info:** Properties, config values, system info.
- **Application Info:** Version, build, commit ID.
- **Loggers:** Dynamic logging level changes at runtime.
- **Tracing:** Request/response tracking for debugging.

🔹 How to Enable Spring Boot Actuator ?

Add dependency (Maven)

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Enable endpoints in application.properties

```
management.endpoints.web.exposure.include=health,info,metrics,env,loggers

management.endpoint.health.show-details=always
```

- By default, only /health and /info are exposed.
- Use include=* to expose all endpoints (not recommended for prod without security).

🔹 Common Endpoints

Endpoint	Description
/actuator/health	Checks app health, DB connection, custom health indicators
/actuator/info	Shows app metadata (version, description, build info)
/actuator/metrics	JVM metrics, memory, CPU usage, HTTP request stats
/actuator/env	Displays environment properties & config values
/actuator/loggers	Shows and modifies logging levels at runtime
/actuator/threaddump	Thread dump for debugging
/actuator/httptrace	Shows last 100 HTTP requests/responses
/actuator/heapdump	Generates a heap dump for memory analysis

🔹 Custom Health Indicator

- **You can add custom checks (e.g., DB, cache, API):**

```
import org.springframework.boot.actuate.health.Health;
import org.springframework.boot.actuate.health.HealthIndicator;
import org.springframework.stereotype.Component;

@Component
public class SweetShopHealthIndicator implements HealthIndicator {

    @Override
    public Health health() {
        boolean sweetsAvailable = checkSweetStock();
        if (sweetsAvailable) {
            return Health.up().withDetail("SweetStock", "Available").build();
        } else {
            return Health.down().withDetail("SweetStock", "Empty").build();
        }
    }
    private boolean checkSweetStock() {
        // logic to check stock
        return true;
    }
}
```

✅ **Access via:**

http://localhost:8080/actuator/health

Output:

```
{
 "status": "UP",
 "details": {
  "SweetStock": "Available"
 }
}
```

🔹 **Interview Points**

1. **Definition:** "Spring Boot Actuator provides production-ready monitoring & management endpoints."

2. **Default Endpoints:** /health, /info

3. **Custom Endpoints:** Can create custom health checks, metrics, or info.

4. **Exposure Control:** Use management.endpoints.web.exposure.include

5. **Security:** Actuator endpoints should be secured in production.

6. **Integration:** Can be integrated with Prometheus, Grafana, Micrometer for monitoring dashboards.

Q:

What is Caching in Spring Boot ?

- "Spring Boot caching stores method results in a cache provider to avoid repeated computation or database calls, using annotations like @Cacheable, @CachePut, and @CacheEvict."

- Supports providers like Ehcache, Redis, Caffeine, Hazelcast, etc.

🔹 **How to Enable Caching ?**

1. **Add spring-boot-starter-cache dependency (optional, Spring Boot auto-configures for some providers).**

2. **Enable caching in your main class:**

```
@SpringBootApplication
@EnableCaching
 public class SweetShopApplication { ... }
```

🔹 Key Annotations

Annotation	Description
@Cacheable	Caches the **result of a method**. If the same parameters are passed again, returns from cache.
@CachePut	Updates the cache without skipping the method execution.
@CacheEvict	Removes data from cache (for updates/deletes).
@Caching	Allows multiple caching operations on a single method.

🔹 Example :

```
@Service
public class SweetService {

    @Autowired
    private SweetRepository sweetRepository;

    @Cacheable(value = "sweetCache", key = "#id")
    public Sweet getSweetById(Long id) {

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

            return sweetRepository.findById(id).orElse(null);
    }

    @CacheEvict(value = "sweetCache", key = "#id")
    public void deleteSweet(Long id) {

        sweetRepository.deleteById(id);
    }

    @CachePut(value = "sweetCache", key = "#sweet.id")
    public Sweet updateSweet(Sweet sweet) {

        return sweetRepository.save(sweet);
    }
}
```

✅ **Behavior:**

- First call to getSweetById(1) → DB hit → cached.
- Second call to getSweetById(1) → returns from cache, no DB hit.
- Updating or deleting invalidates or updates the cache automatically.

🔹 **Common Cache Providers**

Provider	Notes
Ehcache	Simple in-memory caching, easy for small apps
Redis	Distributed caching, persistent and shared across servers
Caffeine	High-performance in-memory cache
Hazelcast	Distributed caching for large-scale apps

🔹 **Interview Points**

1. Definition: "Spring Boot caching stores method results to reduce repeated DB calls or computations."

2. How it works: Enabled via @EnableCaching + annotations like @Cacheable.

3. Annotations: @Cacheable, @CachePut, @CacheEvict, @Caching.

4. Use Case: Fetching DB data, expensive computations, external API calls.

5. Providers: Ehcache, Redis, Caffeine, Hazelcast.

6. Key: Works at method level, unlike Hibernate caching which is entity-level.

THANK YOU !

Ajay Kumar – LinkedIn Profile: https://www.linkedin.com/in/ajay-kumar-sahni-4a02322b8/