JPA REPOSITORY

JPArepositoory is a repository belongs to spring data jpa library which consist of methods already dedclared.

If an entity class consist of parameterized constructor it should also consist default constructor.

Query DSL

DSL means domain specific language

Example: List<Alien> findByname(String name);

It can be findBy or getBy the second word after it should be variable

The first letter of variable should be in capital letter

Should pass the name we are searching for in braces

List<Alien> findBynameOrderByName(String name);

List<Alien> findBynameOrderByNameDesc(String name);

Create own query

Specify query using @Query annotation

Example: @Query("SELECT SUM(e.profit) FROM Emp e")

Double calculateTotalProfit(); //calculate totalProfit using sql query

@Query("SELECT SUM(e.tds) FROM Emp e")

Double calculateTotalTds();

Restful web service /REST API (respresentational state)

Api gives the data that will be in representational state

Every request is state less

Uses inbult method of http protocall

JSON postmappimg and getmapping

@GetMapping(path=”aliens” produces=”{application/xml}”)

Public List<Alien> getAliens()

{

List<Alien> aliens=repo.findAll();

Return aliens;

}

Produces says what server going to responde

@PostMapping(path=”alien”,consumes=”{application/json}”)

Public Alien addAlien(@RequestBody Alien alien)

{

repo.save(alien);

Return alien;

}

We can also write it as

Consumes={MediaType.APPLICATION\_JSON\_VALUE}

Consumes in postmapping specifies what server going to take

We may specify the data type in header

Ex:

key value

Accepts application/xml

Content-Type application/json

In services the service class should be mentioned as @Component to specify as we require spring to give us the object of product service

@GetMapping("/employees")

**public** ResponseEntity<Object> getEmployees() {

// Fetch the list of employees

List<emp> employees = employeeRepository.findAll(); //retrieve employees by employee list

**if**(employees==**null**||employees.isEmpty()) {

**return** ResponseEntity.*status*(404).body("no employee found");

}

// Calculate total profit and TDS

Double totalProfit = employeeRepository.calculateTotalProfit(); //get totalprofit by employeeRepository

Double totalTds = employeeRepository.calculateTotalTds(); //get totalTDS by employeeRepository

// Create a response object to hold employees and calculated values

EmployeeDataResponse response = **new** EmployeeDataResponse(employees, totalProfit, totalTds);

**return** ResponseEntity.*ok*(response); // Return the response as JSON

}

DATABASE CONNECTION AND DEPENDENCIES

When we add additional dependencies that we are not using mainly like jdbc based dependencies such as data jpa ,mysql and others.

When we are not using databse these depndeicies will return errors as they autoconfigure the database connectivivty so when we are not using the database we should enable the database connectivivty byv using code

spring.autoconfigure.exclude=org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration

**TO CALCULATE THE CURRENT YEAR**

**int currentYear = Year.*now*().getValue();**

@Component

It is used to create an object of the class in spring boot.

Default name of an object will be its class name where first letter will be small letter.

When we want to change the name of the object we can change using @Component(“ “)

Keyword.

@Qualifier

as we are using @Qualifier which will search an object using its name so as we changed object name in component we have to change the same object name in qualifier too.

@RequestMapping

Used to handle the functionality of the method it says that whenever I get this request I want to execute or to handle set of code written in that method.

@RequestMapping(“home”)

Public void home(){

}

@ResponseBody

It says whatever we are returning in a set of code in return keyword is the data It will not be the page we are returning it will be the data that displayed on web pages.

@Primary

it states that class or the bean that specified by the @Primary annotation that

will have more priority

@Profile

The @Profile annotation in Spring is used to define beans that should only be registered in the application context for specific **profiles**. Profiles allow you to activate different configurations for different environments without changing the codebase.

@Configuration

@Profile("dev") // Only loaded in the 'dev' profile

public class DevDataSourceConfig {

@Bean

public DataSource dataSource() {

// Configure an H2 in-memory database

return DataSourceBuilder.create()

.url("jdbc:h2:mem:devdb")

.username("sa")

.password("")

.build();

}

}

Webapp

In springboot web app is a folder which is public.

Application.properties

Used to manually configure the data

Prefix

Path of the page is defined with prefix.

Spring.mvc.view.prefix=/pages/

Suffix

Extension of the page is defined with suffix.

Spring.mvc.view.suffix=.jsp

HttpServletRequest

It creates an object that used to take value from user through UI and that object useds to store and print the data inserted through user.

Public String Home(HttpServletRequest req)

Session

When we are using servlet dispatcher two servlet or one servlet and one jsp shares same object at that time we can use session to pass request values when creating session

HttpSession session=req(object name).getSession();

To add the session object

Session.setAttribute(“name”(key),value(value)); //its in key value pairs

HttpServletResponse

Used to handle or to control response of the page as it is same as HttpServletRequest

Public String Home(HttpServletRequest req, HttpServletResponse res)

We can also use

Public String home(String myName,HttpSession session)

Whenever we have myname in our program and we don’t what that to display in url rather than that we want to display name here we can use

Public String home(@RequestParam(“name”) String myName,httpSession session)

This says that whenever name appears in client side request assign that value to the myName variable.

We can call the data as model and also the user requires view the combination of both is Model and View

Model and View

Public ModelAndView home(@RequestParam(“name”) String myName){

ModelAndView mv=new ModelAndView();

mv.addObject(“name”,myName); //key and variable that holds the value

mv.setViewName(“home”); //page name mentioned that can be displayed

return mv;

we can also write it as

Public ModelAndView home(Alien alien){

ModelAndView mv=new ModelAndView();

mv.addObject(“obj”,alien); //key and variable that holds the value

mv.setViewName(“home”); //page name mentioned that can be displayed

return mv;

whenever working with jsp we have to add jasper dependency in pom.xml

H2 database connection

Spring.h2.console.enabled=true

Spring.datasource.platform=h2

Spring.datasource.url=jdbc:h2:mem:navin(database name)

To access h2 databse

Spring/h2:console

Provide a login page should not change username and password just to observe it and remember it you can test connection and after getting test successful click on connect and will get the database by specifying @Entity and @Id in entity class it will create table and columns in database.

Insert

we can create a separate file to insert value to the tables called data.sql and open it by text form :- Insert into alien values(101,’navin’);

Creating repository to perform functionalities with server side

We can extend repository using CrudRepository or JpaRepositories have to mention class name and type of primary key. using @Autowired to instantiate object of repository.

addAlien through UI

Public String addAlien(Alien alien) function to add data to database

repo.save(alien); used to update the inserted data into the database. repo is an object of alienRepo repository used to save the data in database.

fetch alien values from database through UI

public String getAlien(@RequestParam String aid) fuction to print values stored in database.

getAlien(@RequestParam String aid){

ModelAndView mv=new ModelAndView(“showAlien.jsp”);

Alien alien=repo.findById(aid).orElse(new Alien()); //not a good practice

mv.addobject(alien);

Query methods

We can write our own method in spring boot using query language. Requirement is to start method name with findBy or with getBy methods.should end by property in @Entity class.in repository

List <Alien> findByTech(String tech);

List <Alien> findByAidGreaterThan(100 (value));

@Query(“from alien where tech=?1 orderby aname”)

List<Alien> findByTechSorted(String tech);

Controller

System.out.println(repo.findByTech(“java”));

System.out.println(repo.findByAidGreaterThan(100)); System.out.println(repo.findByTechSorted(“java”)); //01:55:01 //09-12-2024

DIFFERENT TYPES OF MAPPINGS IN SPRING BOOT

@GetMapping (Maps HTTP GET requests to a specific method.)

It is used to map a function that used to fetch a particular data by returning a particular page or by returning the data in different formats.

**Use Case**:

* Retrieve data from the server (read-only).
* Render pages or return resources like JSON or HTML.

**Example**:

@GetMapping("/employees")

public List<Employee> getAllEmployees() {

return employeeService.getAllEmployees();

}

@PostMapping (Maps HTTP POST requests to a specific method)

It is used to map the function that used to post some values to the function by user to do different calculations or functions with data in database.

**Use Case**:

* Submit data to the server for processing.
* Create new resources or perform form submissions.

**Example**:

@PostMapping("/employees")

public String addEmployee(@RequestBody Employee employee) {

employeeService.addEmployee(employee);

return "Employee added successfully!";

}

@PutMapping (Maps HTTP PUT requests to a specific method)

It is used to map a function that performs actions as adding new or updating existing data in specified database by user through different page.

**Use Case**:

* Update an existing resource.
* Typically used for replacing an entire resource.

**Example**:

@PutMapping("/employees/{id}")

public String updateEmployee(@PathVariable Long id, @RequestBody Employee employee) {

employeeService.updateEmployee(id, employee);

return "Employee updated successfully!";

}

@PatchMapping

* **Description**: Maps HTTP PATCH requests to a specific method.
* **Use Case**:
  + Partially update an existing resource.
  + More efficient than PUT if only a subset of fields needs to be updated.

**Example**:

@PatchMapping("/employees/{id}")

public String updateEmployeeField(@PathVariable Long id, @RequestBody Map<String, Object> updates) {

employeeService.updateEmployeePartial(id, updates);

return "Employee updated successfully!";

}

@DeleteMapping (Maps HTTP DELETE requests to a specific method)

It is used to map a function that performs actions as deleting the existing data from database by taking the values of that particular data by the user through different page.

* **Use Case**:
  + Delete a resource from the server.

**Example**:

@DeleteMapping("/employees/{id}")

public String deleteEmployee(@PathVariable Long id) {

employeeService.deleteEmployee(id);

return "Employee deleted successfully!";

}

@RequestMapping

* **Description**: A versatile annotation that can map HTTP requests to specific methods for multiple HTTP methods (GET, POST, etc.).
* **Attributes**:
  + method: Specifies the HTTP method (e.g., RequestMethod.GET).
  + path or value: Specifies the URL path.

**Example**:

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

public List<Employee> getAllEmployees() {

return employeeService.getAllEmployees();

}

@CrossOrigin

* **Description**: Enables Cross-Origin Resource Sharing (CORS) for a specific endpoint.
* **Use Case**:
  + Allow requests from external domains.

**Example**:

@CrossOrigin(origins = "http://example.com")@GetMapping("/employees")

@ExceptionHandler **(Special Mapping)**

* **Description**: Maps exceptions to specific handler methods.
* **Use Case**:
  + Handle exceptions gracefully and provide custom error responses.

**Example**:

@ExceptionHandler(ResourceNotFoundException.class)

public ResponseEntity<String> handleNotFoundException(ResourceNotFoundException ex) {

return ResponseEntity.status(HttpStatus.NOT\_FOUND).body(ex.getMessage());

}

### ****Key Differences and Use Cases****

| **Mapping Type** | **HTTP Verb** | **Use Case** | **Example** |
| --- | --- | --- | --- |
| @GetMapping | GET | Fetch data, load pages | GET /employees |
| @PostMapping | POST | Submit data, create resources | POST /employees |
| @PutMapping | PUT | Update entire resource | PUT /employees/{id} |
| @PatchMapping | PATCH | Partially update a resource | PATCH /employees/{id} |
| @DeleteMapping | DELETE | Delete a resource | DELETE /employees/{id} |
| @RequestMapping | Any | Flexible, define both verb and path | RequestMethod.GET /employees |
| @ExceptionHandler | Any | Handle exceptions | Custom error handling |

Token generation and validating and storing redirecting

Response 1

To store the token in cookies using the provided code, you can modify the login endpoint to include setting the token as a cookie in the HTTP response. Here's how you can adjust your login method:

**Modified login Method with Cookie Support**

java

Copy code

@PostMapping("/login")

public ResponseEntity<Object> login(@RequestParam String username,

@RequestParam String password,

HttpServletResponse response) {

User user = userRepository.findByUsername(username); // Find the user by username

if (user != null && user.getPassword().equals(password)) {

// Generate a JWT token for the user

String token = tokenService.generateToken(username);

// Create a cookie to store the token

Cookie jwtCookie = new Cookie("jwt", token);

jwtCookie.setHttpOnly(true); // Prevent JavaScript access (helps mitigate XSS attacks)

jwtCookie.setMaxAge(3600); // Set cookie expiration to 1 hour

jwtCookie.setPath("/"); // Make cookie available to all endpoints in the domain

response.addCookie(jwtCookie); // Add cookie to the response

return ResponseEntity.ok(new AuthResponse("Login Successful", username, null, token));

} else {

return ResponseEntity.status(401).body("Invalid credentials");

}

}

**Explanation of Changes**

1. **Cookie jwtCookie**:
   * A new cookie is created with the token value.
   * The cookie name is "jwt", and its value is the JWT generated by the TokenService.
2. **setHttpOnly(true)**:
   * Makes the cookie inaccessible to client-side JavaScript, enhancing security against XSS attacks.
3. **setMaxAge(3600)**:
   * Sets the cookie's expiration time to 1 hour (3600 seconds). Match this with your token's expiration time.
4. **setPath("/")**:
   * Ensures the cookie is available across all endpoints of your application.
5. **response.addCookie(jwtCookie)**:
   * Adds the cookie to the HTTP response, sending it to the client.

**Validating Token from Cookies**

To validate the token stored in cookies, retrieve it from the HttpServletRequest in your secured endpoints:

java

Copy code

@GetMapping("/employees")

public ResponseEntity<Object> getEmployees(@CookieValue(value = "jwt", required = false) String token) {

if (token == null || !tokenService.isTokenValid(token)) {

return ResponseEntity.status(401).body("Invalid or expired token");

}

// Fetch employee data

List<emp> employees = employeeRepository.findAll();

Double totalProfit = employeeRepository.calculateTotalProfit();

Double totalTds = employeeRepository.calculateTotalTds();

// Create response object

EmployeeDataResponse response = new EmployeeDataResponse(employees, totalProfit, totalTds);

return ResponseEntity.ok(response);

}

**How It Works**

1. **Retrieve the Token from the Cookie**:
   * Use the @CookieValue annotation to access the "jwt" cookie value directly in the request handler.
2. **Validate the Token**:
   * Pass the token to tokenService.isTokenValid() to ensure it's not expired or tampered with.
3. **Handle Authorization**:
   * If the token is invalid, return a 401 Unauthorized response.
   * If valid, proceed to fetch and return the requested data.

**Logout Implementation**

To log out the user, clear the token by setting the cookie with a max age of 0:

java

Copy code

@PostMapping("/logout")

public ResponseEntity<String> logout(HttpServletResponse response) {

Cookie jwtCookie = new Cookie("jwt", null);

jwtCookie.setHttpOnly(true);

jwtCookie.setMaxAge(0); // Expire the cookie immediately

jwtCookie.setPath("/");

response.addCookie(jwtCookie);

return ResponseEntity.ok("Logged out successfully");

}

**Benefits of Storing Tokens in Cookies**

* **Secure**: By setting HttpOnly, the token is protected from JavaScript-based attacks.
* **Automatic Inclusion**: Browsers automatically send cookies with each request, simplifying client implementation.
* **Cross-Origin Protection**: If Secure and SameSite attributes are configured correctly, cookies are protected from CSRF attacks.

Let me know if you'd like help integrating these changes!

I prefer this response

ChatGPT

Response 2

In the provided code, the token is returned in the **response body** as part of the AuthResponse object. To store the token in cookies, you'll need to modify the code slightly. Here's how you can do it:

**Steps to Store the Token in Cookies**

**1. Modify the Login Endpoint to Set the Token in a Cookie**

Instead of just returning the token in the response body, add it to a cookie. Here's the updated code:

java

Copy code

@PostMapping("/login")

public ResponseEntity<Object> login(@RequestParam String username,

@RequestParam String password,

HttpServletResponse response) {

User user = userRepository.findByUsername(username); // Find the user by username

if (user != null && user.getPassword().equals(password)) {

// Generate a JWT token for the user

String token = tokenService.generateToken(username);

// Add the token to an HttpOnly cookie

Cookie jwtCookie = new Cookie("jwt", token);

jwtCookie.setHttpOnly(true); // Prevent JavaScript access to the cookie (mitigates XSS attacks)

jwtCookie.setSecure(true); // Use this in production to ensure the cookie is sent only over HTTPS

jwtCookie.setMaxAge(3600); // Set expiration to 1 hour (in seconds)

jwtCookie.setPath("/"); // Cookie is valid for the entire domain

response.addCookie(jwtCookie);

return ResponseEntity.ok("Login successful");

} else {

return ResponseEntity.status(401).body("Invalid credentials");

}

}

**2. Validate the Token from Cookies in the Protected Endpoints**

In the /employees endpoint, extract the token from the cookies instead of relying on the Authorization header.

java

Copy code

@GetMapping("/employees")

public ResponseEntity<Object> getEmployees(@CookieValue(value = "jwt", required = false) String token) {

if (token == null || !tokenService.isTokenValid(token)) {

return ResponseEntity.status(401).body("Invalid or expired token");

}

try {

// Fetch the list of employees

List<emp> employees = employeeRepository.findAll();

if (employees == null || employees.isEmpty()) {

return ResponseEntity.status(404).body("No employees found");

}

// Calculate total profit and TDS

Double totalProfit = employeeRepository.calculateTotalProfit();

Double totalTds = employeeRepository.calculateTotalTds();

// Create response object

EmployeeDataResponse response = new EmployeeDataResponse(employees, totalProfit, totalTds);

return ResponseEntity.ok(response);

} catch (Exception e) {

e.printStackTrace();

return ResponseEntity.status(500).body("Internal server error");

}

}

**Key Points**

1. **Set Cookie in the Login Endpoint**:
   * The token is stored in a secure, HTTP-only cookie.
   * Use the @CookieValue annotation to read the token from the cookie in subsequent requests.
2. **Advantages of Using Cookies**:
   * Cookies automatically travel with requests to the server (for the same domain), so the frontend does not need to explicitly send the token in every request.
   * HTTP-only cookies prevent JavaScript access, making them more secure against XSS attacks.
3. **Security Considerations**:
   * **Secure Cookies**: Always set secure=true to ensure cookies are only sent over HTTPS.
   * **SameSite Attribute**: Consider using the SameSite attribute to prevent cross-site request forgery (CSRF) attacks:

java

Copy code

jwtCookie.setSecure(true);

jwtCookie.setHttpOnly(true);

jwtCookie.setPath("/");

jwtCookie.setMaxAge(3600);

jwtCookie.setAttribute("SameSite", "Strict");

1. **Logout Endpoint**: To allow users to log out, clear the cookie by setting its max age to 0.

java

Copy code

@PostMapping("/logout")

public ResponseEntity<String> logout(HttpServletResponse response) {

Cookie jwtCookie = new Cookie("jwt", null);

jwtCookie.setHttpOnly(true);

jwtCookie.setSecure(true);

jwtCookie.setMaxAge(0); // Delete the cookie

jwtCookie.setPath("/");

response.addCookie(jwtCookie);

return ResponseEntity.ok("Logged out successfully");

}

This setup ensures that tokens are securely stored in cookies and are validated with every request, adhering to secure API practices. Let me know if you need further assistance!

I prefer this response

Top of Form

@PostConstruct

@PostConstruct is an annotation in Java used to mark a method that should be executed **immediately after the bean's initialization** (i.e., after dependency injection is complete and the bean is fully constructed). It comes from the **Java EE specification** (specifically the javax.annotation package) but is also supported in Spring Framework.

When a method is annotated with @PostConstruct, Spring ensures that this method is invoked **once** during the bean's lifecycle, right after the bean is initialized and its dependencies are injected.

 Spring creates the bean instance.

 Spring injects all the required dependencies into the bean.

 Spring calls the @PostConstruct annotated method (before the bean is available for use by other beans).

**Key Points**

1. **Method Requirements:**
   * The method annotated with @PostConstruct must:
     + Be void.
     + Take no arguments.
     + Not throw a checked exception.
2. **Alternative in Spring:**
   * Instead of @PostConstruct, Spring provides an alternative with the InitializingBean interface or the @Bean(initMethod) attribute in Java-based configuration. However, @PostConstruct is more concise and commonly used.

Example with InitializingBean:

java

Copy code

@Component

public class ExampleBean implements InitializingBean {

@Override

public void afterPropertiesSet() throws Exception {

System.out.println("afterPropertiesSet called!");

}

}

1. **Lifecycle Management:**
   * If you're using @PostConstruct, you should also be aware of @PreDestroy, which is used to clean up resources before the bean is destroyed.

Spring boot Pagination and sorting using Spring DataJPA

@PostConstruct

Acts as init method here used to insert 200 random values in specified database

**Example:**

@PostConstruct

Public void initDB(){

List<product> products=intStream.rangeClosed(1,200)

.mapToObj(i 🡪 new Product(“product” +I, new Random().nextInt(100), new

Random().nextInt(50000)))

.collect(Collectors.toList());

Repository.saveAll(products);

}

**Create method to return values sorted in service class**

Public list<product> findproductswithSorting(String field){

return repository.findAll(Sort.by(Sort.Direction ASC,field));

}

**Call the method in controller class**

@GetMapping(“/{field}”)

private APIResponse<List <Product>> getProductwithSort(@PathVariable String field){

List <product> allProducts=service. findproductswithSorting(field);

return new APIResponse<>(allProducts.size,allProducts);

}

**Create method to perform pagination through backend**

public Page<Product> findProductwithPagination(int offset,int pageSize){

page <product> products=repository.findAll(pageRequest.of(offset,pageSize));

return products;

}

**Call method created in service class through controller**

@GetMapping(“/pagination/{offset}/{pageSize}”)

private APIResponse<List <Product>> getProductwithSort(@PathVariable int offset,@PathVariable int pageSize){

Page <product> productsWithPagination=service. findProductwithPagination(offset,pageSize);

return new APIResponse<>(productsWithPagination.getSize(),productsWithPagination);

}

Create method that can perform both pagination and also the sorting

public Page<Product> findProductwithPaginationAndSorting(int offset,int pageSize,String field){

page <product> products=repository.findAll(pageRequest.of(offset,pageSize).withSort(Sort.by(filed));

return products;

}

Validation in entity class And ExceptionHandler for handling exception

@Data

@AllArgsConstructor(staticName=”build”)

@NoArgsConstructor

public class UserRequest{

@NotNull(message=”username should not be null”) //javax validation.contraints

private String name;

@Email(message=”invalid email address”)

private String email;

@Pattern(regexp= “^\\d{10}$”,message=”invalid mbile number entered”)

private String mobile;

private String gender;

@Min(18)

@Max(60)

private int age;

@NotBlank //together considers not null as well as not empty

private String nationality;

**creating exception handler class in advice package**

@RestControllerAdvice

public class ApplicationExceptionHandler{

@ResponseStatus(HttpStatus.BAD\_REQUEST)

@ExceptionHandler(MethodAurgumentNotValidException.class)

public Map<String,String> handleInvalidAurgument(MethodAurgumentNotValidException ex){

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

ex.getBindingResult().getFieldErrors().forEach(error ->{

errorMap.put(error.getField(),error.getDefaultmessage());

});

return errorMap;

}

@ResponseStatus(HttpStatus.INTERNAL\_SERVER\_ERROR)

@ExceptionHandler(userNotFoundException.class)

public Map<String,String> handleBusinessException(UserNotFoundException ex) {

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

errorMap.put(“errorMessage”,ex.getMessage());

}

**Handling exception when the error for accessing invalid id is not shown**

Public user getUser(int id){

User user=repository.findByUserId(id);

If(user!=null){

return user;

}else{

Throw new UserNotFoundExcedption(“user not found with id”+id);

}

}

**Create new package as exception create class as UserNotFoundException**

Public class userNotFoundException extends Exception{

Public UserNotFoundException(String message){

Super(message);

}

}

**Go to controller and forcefully throw the eexception in controller**

@GetMapping(“/{id}”)

Public ResponseEntity<user> getUser(@PathVariable int id)throws UserNotFoundException{

return responseEntity.ok(service.getuser(id));

}

**Using @Valid in controller for validation**

In **Spring Boot** and **Spring Data JPA**, the @Valid annotation is used to trigger validation for the fields of a request object (or model) based on validation constraints specified by annotations like @NotNull, @Size, @Min, etc. It is commonly used in combination with the **javax.validation** or **jakarta.validation** API.

Steps to Use @Valid for Validation in Spring Boot

### 1. Add Required Dependencies

Ensure you have the **Validation API** and **Hibernate Validator** dependencies in your pom.xml:

xml

Copy code

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-validation</artifactId>

</dependency>

This dependency includes **Hibernate Validator**, which is the default implementation of the **Java Bean Validation API** (JSR-380).

### 2. Add Validation Annotations in Your Entity or DTO

You can add validation constraints to your class fields using annotations like @NotNull, @Size, @Min, @Max, etc.

**Example DTO:**

java

Copy code

import jakarta.validation.constraints.NotNull;

import jakarta.validation.constraints.Size;

import jakarta.validation.constraints.Min;

public class EmployeeDTO {

@NotNull(message = "Name must not be null")

@Size(min = 3, max = 50, message = "Name must be between 3 and 50 characters")

private String name;

@NotNull(message = "Salary is required")

@Min(value = 1, message = "Salary must be greater than 0")

private Double salary;

@NotNull(message = "Department is required")

private String department;

// Getters and Setters

}

### 3. Use @Valid in the Controller Layer

Use @Valid on the method parameter to trigger validation when the request data is received.

**Example Controller:**

java

Copy code

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import jakarta.validation.Valid;

@RestController

@RequestMapping("/api/employees")

public class EmployeeController {

@PostMapping("/add")

public ResponseEntity<String> addEmployee(@Valid @RequestBody EmployeeDTO employeeDTO) {

// If validation passes, this code executes

return ResponseEntity.ok("Employee added successfully");

}

}

### ****Explanation****:

* @Valid: Tells Spring Boot to validate the EmployeeDTO object before entering the method.
* @RequestBody: Maps the JSON request body to the EmployeeDTO.

If validation fails, Spring Boot will return an error response with a **400 Bad Request** status and a list of validation error message.

### 4. Handle Validation Errors Using @ExceptionHandler

By default, Spring Boot returns an error response with the validation error details. However, you can customize the response using a **@ControllerAdvice** to handle exceptions.

**Example Global Exception Handler**:

java

Copy code

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.MethodArgumentNotValidException;

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

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

import java.util.HashMap;

import java.util.Map;

@ControllerAdvice

public class GlobalExceptionHandler {

@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);

}

}

### ****Explanation****:

* **@ControllerAdvice**: Enables global exception handling for your application.
* **@ExceptionHandler**: Captures specific exceptions (MethodArgumentNotValidException) triggered by validation failures.
* The method formats and returns validation errors in a user-friendly format.

### 5. Testing the Validation

Assume you send an invalid request body to the endpoint /api/employees/add:

**Invalid Request JSON**:

json

Copy code

{

"name": "",

"salary": -10,

"department": null

}

**Response**:

json

Copy code

{

"name": "Name must be between 3 and 50 characters",

"salary": "Salary must be greater than 0",

"department": "Department is required"

}

Spring Boot automatically validates the request using the constraints and returns the custom error messages.

**Common Validation Annotations**

| **Annotation** | **Description** |
| --- | --- |
| @NotNull | Field must not be null. |
| @Size(min, max) | Field length must be between min and max. |
| @Min(value) | Field value must be greater than or equal to value. |
| @Max(value) | Field value must be less than or equal to value. |
| @NotBlank | Field must not be null and must contain characters. |
| @NotEmpty | Field must not be null or empty (size > 0). |
| @Pattern(regex) | Field must match the provided regular expression. |
| @Email | Field must be a valid email address. |

**Summary**

1. Add **spring-boot-starter-validation** dependency.
2. Use annotations like @NotNull, @Size, @Min, etc., in your DTO or Entity classes.
3. Use @Valid in the controller to validate incoming requests.
4. Customize validation error responses using @ControllerAdvice.

Crud operations in JPA repository

In Spring Data JPA, the JpaRepository interface provides built-in methods for performing CRUD (Create, Read, Update, Delete) operations. It simplifies database interactions without needing boilerplate code.

Here’s an explanation of each CRUD operation:

**1. Create (Save data into the database)**

* The save() method in the JpaRepository is used to insert or update data in the database.

Usage**:**

java

Copy code

Employee emp = new Employee();

emp.setName("John Doe");

emp.setPosition("Developer");

// Save new employee record

employeeRepository.save(emp);

* If the ID of the entity does not exist, it will insert the record.
* If the ID exists, it will update the existing record.

**2. Read (Retrieve data from the database)**

JPA provides multiple methods to read data:

* Retrieve all records.
* Retrieve a record by its ID.
* Retrieve records based on conditions.

**Built-in methods:**

* **findAll()** → Retrieves all records in the table.

java

Copy code

List<Employee> employees = employeeRepository.findAll();

* **findById()** → Retrieves a single record by its primary key.

java

Copy code

Optional<Employee> emp = employeeRepository.findById(1L); // ID = 1

emp.ifPresent(e -> System.out.println(e.getName()));

* **Custom Query Methods** → You can define methods based on entity fields:  
  Example:

java

Copy code

List<Employee> findByName(String name);

**Example:**

java

Copy code

List<Employee> devs = employeeRepository.findByPosition("Developer");

**3. Update (Modify existing data)**

* The save() method also performs update operations.
* If the entity’s primary key already exists in the database, JPA will update the corresponding record.

**Usage:**

java

Copy code

Optional<Employee> empOptional = employeeRepository.findById(1L);

if (empOptional.isPresent()) {

Employee emp = empOptional.get();

emp.setPosition("Senior Developer"); // Update field

employeeRepository.save(emp); // Save updated entity

}

**4. Delete (Remove data from the database)**

JPA provides the following methods for deleting records:

* **deleteById()** → Deletes a record by its ID.

java

Copy code

employeeRepository.deleteById(1L); // Deletes the record with ID = 1

* **delete()** → Deletes a specific entity.

java

Copy code

Employee emp = new Employee();

emp.setId(1L);

employeeRepository.delete(emp); // Deletes the record with matching ID

* **deleteAll()** → Deletes all records from the table.

java

Copy code

employeeRepository.deleteAll();

**Summary of Key JPA CRUD Methods**

| **Method** | **Purpose** |
| --- | --- |
| save(entity) | Insert or update an entity. |
| findById(id) | Retrieve an entity by ID. |
| findAll() | Retrieve all entities. |
| deleteById(id) | Delete an entity by ID. |
| delete(entity) | Delete a specific entity. |
| deleteAll() | Delete all entities. |
| existsById(id) | Check if an entity exists. |
| count() | Count the total entities. |

**Custom Queries in JPA Repository**

Sometimes, the built-in CRUD methods may not be enough. You can define custom query methods in your repository:

**Example:**

java

Copy code

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

List<Employee> findByPosition(String position); // Find all employees by position

@Query("SELECT e FROM Employee e WHERE e.name = :name")

List<Employee> findByName(@Param("name") String name);

}

**SpringBoot DataJPA Relationships: @OneToMany, @ManyToOne, @ManyToMany, @OneToOne**

In **Spring Data JPA**, relationships define how entities are connected to each other in a relational database. These relationships are represented using annotations like @OneToOne, @OneToMany, @ManyToOne, and @ManyToMany. They correspond to the relationships in the database (primary key, foreign key constraints).

Here is a detailed explanation of each relationship with examples:

**1. @OneToOne Relationship**

A **OneToOne** relationship is when one entity is associated with exactly **one other entity**.

### ****Use case example****:

An **Employee** has exactly **one Address**.

### ****Implementation****:

java

Copy code

@Entity

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

@OneToOne(cascade = CascadeType.ALL)

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

private Address address;

// Getters and Setters

}

@Entity

public class Address {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String city;

private String state;

// Getters and Setters

}

### ****Explanation****:

* **@OneToOne**: Defines a one-to-one relationship.
* **cascade = CascadeType.ALL**: Changes in the parent entity are propagated to the child (e.g., save or delete).
* **@JoinColumn**: Specifies the foreign key column (e.g., address\_id) in the Employee table.

**2. @OneToMany Relationship**

A **OneToMany** relationship is when one entity is associated with **multiple child entities**.

### ****Use case example****:

A **Department** has multiple **Employees**.

### ****Implementation****:

java

Copy code

@Entity

public class Department {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

@OneToMany(mappedBy = "department", cascade = CascadeType.ALL)

private List<Employee> employees;

// Getters and Setters

}

@Entity

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

@ManyToOne

@JoinColumn(name = "department\_id")

private Department department;

// Getters and Setters

}

### ****Explanation****:

* **@OneToMany**: A department can have multiple employees.
* **mappedBy**: Specifies that the department field in Employee owns the relationship.
* **@ManyToOne**: In the child entity (Employee), the relationship is defined as ManyToOne since multiple employees can belong to one department.
* **@JoinColumn**: Defines the foreign key column in the Employee table.

**3. @ManyToOne Relationship**

The **ManyToOne** relationship means **many entities** are associated with **one parent entity**.

### ****Use case example****:

Many **Employees** belong to one **Department** (reverse of @OneToMany).

### ****Implementation****:

As shown above, in the Employee entity:

java

Copy code

@ManyToOne

@JoinColumn(name = "department\_id")

private Department department;

### ****Explanation****:

* **@ManyToOne**: Defines the relationship where many employees belong to one department.
* This is the owning side of the relationship.

**4. @ManyToMany Relationship**

A **ManyToMany** relationship means multiple entities can be associated with multiple other entities.

### ****Use case example****:

A **Student** can enroll in multiple **Courses**, and each **Course** can have multiple **Students**.

### ****Implementation****:

java

Copy code

@Entity

public class Student {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

@ManyToMany

@JoinTable(

name = "student\_course",

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

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

)

private List<Course> courses;

// Getters and Setters

}

@Entity

public class Course {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String title;

@ManyToMany(mappedBy = "courses")

private List<Student> students;

// Getters and Setters

}

### ****Explanation****:

* **@ManyToMany**: Defines a many-to-many relationship.
* **@JoinTable**: Specifies the join table (e.g., student\_course) to link both entities.
  + joinColumns → Specifies the foreign key for the **Student** table.
  + inverseJoinColumns → Specifies the foreign key for the **Course** table.
* **mappedBy**: On the reverse side (Course), the relationship is managed by the courses field in Student.

**Key Differences Between Relationships**

| **Annotation** | **Relationship** | **Description** |
| --- | --- | --- |
| @OneToOne | 1-to-1 | One entity is associated with one other. |
| @OneToMany | 1-to-Many | One entity has many child entities. |
| @ManyToOne | Many-to-1 | Many entities are related to one parent. |
| @ManyToMany | Many-to-Many | Many entities are related to many others. |

**Cascade Types**

When defining relationships, cascade options control how operations (e.g., save, delete) are propagated from the parent to child entities.

| **Cascade Option** | **Description** |
| --- | --- |
| CascadeType.PERSIST | Save operation propagates to the child. |
| CascadeType.MERGE | Merge (update) operation propagates. |
| CascadeType.REMOVE | Delete operation propagates to the child. |
| CascadeType.ALL | All operations (save, update, delete) propagate. |
| CascadeType.DETACH | Detach operation propagates. |
| CascadeType.REFRESH | Refresh operation propagates. |

**Summary**

* **@OneToOne**: One entity has one related entity.
* **@OneToMany** and **@ManyToOne**: One entity relates to many, or many entities relate to one.
* **@ManyToMany**: Many entities relate to many entities.
* Use cascade for propagating operations.
* Use mappedBy to indicate the non-owning side of the relationship.

**Spring Boot Security Role-Based Access Control (RBAC).**

**Spring Boot Role-Based Access Control (RBAC)**

Role-Based Access Control (RBAC) is a mechanism for managing user permissions based on their roles. In Spring Boot, RBAC can be implemented easily with Spring Security to control access to different resources or endpoints based on roles assigned to users.

**Key Concepts in RBAC**

Roles: Represent a group of permissions (e.g., ADMIN, USER, MANAGER).

Permissions: Define specific actions a user can perform (e.g., READ, WRITE, DELETE).

Users: Entities (e.g., people or systems) who are assigned one or more roles.

How RBAC Works in Spring Boot?

User Authentication: Users are first authenticated to confirm their identity (via username/password, JWT tokens, etc.).

Role Assignment: Users are assigned specific roles (e.g., ADMIN, USER).

Authorization: Based on the role, users are granted or denied access to endpoints/resources.

Steps to Implement RBAC in Spring Boot

**1. Add Spring Security Dependencies**

Add Spring Security dependencies to your project’s pom.xml file:

xml

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-thymeleaf</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.security</groupId>

<artifactId>spring-security-test</artifactId>

</dependency>

**2. Configure Users and Roles**

You need to define users and roles. You can use:

In-Memory Authentication (for testing/demo purposes).

Database Authentication (real-world applications).

**Option 1: In-Memory Authentication**

Here’s an example where users and roles are defined directly in the configuration:

java

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

import org.springframework.security.crypto.password.PasswordEncoder;

import org.springframework.security.web.SecurityFilterChain;

@Configuration

public class SecurityConfig {

@Bean

public PasswordEncoder passwordEncoder() {

return new BCryptPasswordEncoder();

}

@Bean

public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

http

.authorizeHttpRequests(authz -> authz

.antMatchers("/admin/\*\*").hasRole("ADMIN") // Only ADMIN role can access /admin/\*\*

.antMatchers("/user/\*\*").hasAnyRole("USER", "ADMIN") // USER or ADMIN roles

.antMatchers("/public/\*\*").permitAll() // Public endpoints accessible by anyone

.anyRequest().authenticated() // All other endpoints require authentication

)

.formLogin(form -> form

.loginPage("/login") // Custom login page

.permitAll()

)

.logout(logout -> logout.permitAll());

return http.build();

}

protected void configure(AuthenticationManagerBuilder auth) throws Exception {

auth

.inMemoryAuthentication()

.withUser("admin").password(passwordEncoder().encode("admin123")).roles("ADMIN")

.and()

.withUser("user").password(passwordEncoder().encode("user123")).roles("USER");

}

}

**Option 2: Database Authentication**

In real-world applications, users and roles are stored in a database. Here’s an example setup:

Database Tables:

users: Stores user credentials (username, password).

roles: Stores role information (e.g., ADMIN, USER).

user\_roles: Maps users to roles (many-to-many relationship).

SQL Schema Example:

sql

CREATE TABLE users (

id BIGINT AUTO\_INCREMENT PRIMARY KEY,

username VARCHAR(255) UNIQUE NOT NULL,

password VARCHAR(255) NOT NULL,

enabled BOOLEAN NOT NULL

);

CREATE TABLE roles (

id BIGINT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(255) UNIQUE NOT NULL

);

CREATE TABLE user\_roles (

user\_id BIGINT,

role\_id BIGINT,

PRIMARY KEY (user\_id, role\_id),

FOREIGN KEY (user\_id) REFERENCES users(id),

FOREIGN KEY (role\_id) REFERENCES roles(id)

);

**Load User Details: Implement UserDetailsService to fetch users and roles from the database.**

java

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;

import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

import org.springframework.security.crypto.password.PasswordEncoder;

import org.springframework.security.core.userdetails.UserDetailsService;

@Configuration

public class SecurityConfig {

@Autowired

private UserDetailsService userDetailsService;

@Bean

public PasswordEncoder passwordEncoder() {

return new BCryptPasswordEncoder();

}

@Autowired

public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {

auth.userDetailsService(userDetailsService).passwordEncoder(passwordEncoder());

}

}

**Custom UserDetailsService Implementation:**

java

import org.springframework.security.core.userdetails.User;

import org.springframework.security.core.userdetails.UserDetails;

import org.springframework.security.core.userdetails.UserDetailsService;

import org.springframework.security.core.userdetails.UsernameNotFoundException;

import org.springframework.stereotype.Service;

@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 User.builder()

.username(user.getUsername())

.password(user.getPassword())

.roles(user.getRoles().toArray(new String[0]))

.build();

}

}

**3. Role-Based Access Control in Endpoints**

Use @PreAuthorize or @Secured annotations to restrict access at the method level.

java

import org.springframework.security.access.prepost.PreAuthorize;

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

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

@RestController

public class AdminController {

@GetMapping("/admin/dashboard")

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

public String adminDashboard() {

return "Welcome to Admin Dashboard!";

}

@GetMapping("/user/profile")

@PreAuthorize("hasAnyRole('USER', 'ADMIN')")

public String userProfile() {

return "Welcome to User Profile!";

}

}

@PreAuthorize("hasRole('ADMIN')"): Only users with the ADMIN role can access this method.

@PreAuthorize("hasAnyRole('USER', 'ADMIN')"): Users with either USER or ADMIN roles can access the method.

**Key Annotations**

@PreAuthorize: Allows method-level security.

@Secured: Similar to @PreAuthorize but simpler.

@EnableGlobalMethodSecurity: Enables method-level security features.

**RBAC Flow Example**

User tries to access an endpoint.

Spring Security intercepts the request.

User is authenticated (via login).

Spring Security checks the user’s roles.

Based on the roles:

Access is granted if roles match.

Access is denied otherwise.

**Benefits of RBAC**

Centralized Control: Role definitions and access rules are managed centrally.

Granular Access: Allows fine-grained access to resources based on roles.

Flexibility: Easy to assign roles and modify permissions.

If logged in user is ADMIN then can give ADMIN access and MODERATOR(User) access.

If logged in user is MODERATOR(user) then can give MODERATOR access.

**OAuth2 and openid connect concepts in spring boot**

**OAuth 2.0: Authorization Framework**

OAuth 2.0 is a protocol that allows secure authorization for accessing resources on behalf of a user without exposing their credentials (e.g., username and password).

* Purpose: Authorization (delegating access).
* How it works:
  1. A user wants to use a third-party app (client) to access their data on a resource server (e.g., Google Drive, Facebook).
  2. Instead of sharing credentials with the third-party app, the app redirects the user to the resource server (identity provider) for authorization.
  3. After the user grants permission, the resource server issues an **access token** to the client.
  4. The third-party app uses this token to access the user's resources securely.
* Example Use Case: You log into an app using your Google account, and the app requests permission to access your Google Drive files.
* Key Components:
  1. **Resource Owner (User):** The entity granting access to their resources.
  2. **Client:** The third-party app requesting access.
  3. **Authorization Server:** Issues tokens after user consent.
  4. **Resource Server:** Hosts the user’s resources (e.g., APIs like Google Drive).
  5. **Access Token:** A credential used to access protected resources.

**OpenID Connect (OIDC): Authentication Layer**

OpenID Connect is built on top of OAuth 2.0 and adds an **authentication layer**. While OAuth 2.0 is about authorization, OIDC ensures the user is authenticated.

* **Purpose:** Authentication (verifying identity).
* **How it works:**
  1. Like OAuth 2.0, the client app redirects the user to an identity provider.
  2. The identity provider authenticates the user (e.g., checks their username/password or biometrics).
  3. After successful authentication, the identity provider returns an **ID token** to the client.
  4. The client uses the ID token to confirm the user’s identity.
* **Example Use Case:** When you log into an app using "Sign in with Google," OIDC ensures you are who you claim to be.
* **Key Components:**
  1. **ID Token:** A JSON Web Token (JWT) that contains information about the user (e.g., name, email, user ID).
  2. **User Info Endpoint:** Provides additional details about the authenticated user.
  3. **Scopes:** OIDC introduces the openid scope for user authentication.

**Differences Between OAuth 2.0 and OpenID Connect**

| **Aspect** | **OAuth 2.0** | **OpenID Connect** |
| --- | --- | --- |
| **Focus** | Authorization | Authentication |
| **Primary Token** | Access Token | ID Token |
| **Use Case** | Access resources securely | Verify user identity |
| **Example** | Allow an app to access your Google Drive. | Log in to an app with Google credentials. |

In short:

* OAuth 2.0 = "Does the app have permission to access my stuff?"
* OpenID Connect = "Who is the user logging in?"

**CSRF (Cross-Site Request Forgery)**

**What is CSRF?**

**CSRF (Cross-Site Request Forgery)** is a type of attack that tricks a user into performing actions they didn't intend to perform on a web application in which they are authenticated. For example, a malicious site might trick you into unknowingly transferring money or changing your password on a bank website where you are logged in.

**How does CSRF work?**

1. A user logs into a legitimate website (e.g., a banking site) and their session is authenticated.
2. The attacker sends the user a malicious link or script (via email, chat, etc.).
3. If the user clicks the link or executes the script while still logged into the legitimate site, the attack will use the user's session to perform unauthorized actions on their behalf.

**CSRF Protection in Spring Boot**

Spring Security provides built-in CSRF protection by default to mitigate this risk.

**How CSRF Protection Works:**

1. **CSRF Token Generation:**
   * When a user accesses a form in the application, Spring Security generates a unique **CSRF token** for their session.
   * This token is embedded in forms or sent as part of requests (e.g., as a hidden input field or request header).
2. **CSRF Token Validation:**
   * When the user submits the form, the CSRF token is sent along with the request.
   * The server checks if the token is valid and matches the one stored in the session.
   * If the token is invalid or missing, the server rejects the request.
3. **Why It Works:**
   * The attacker cannot know the user's CSRF token because it is tied to the user's session and not shared across domains.

**Enabling CSRF Protection in Spring Boot**

CSRF protection is **enabled by default** in Spring Security. You don’t need to explicitly turn it on unless you’ve disabled it.

**Disabling CSRF Protection (Not Recommended)**

In certain cases (e.g., APIs that don’t use cookies for authentication), you might disable CSRF as follows:

java

Copy code

@Configuration

@EnableWebSecurity

public class SecurityConfig extends WebSecurityConfigurerAdapter {

@Override

protected void configure(HttpSecurity http) throws Exception {

http.csrf().disable(); // Disables CSRF protection

}

}

**Using CSRF Tokens in Forms**

If you’re building a web application with forms, you need to include the CSRF token. Spring Security provides helper functions to do this.

1. **In Thymeleaf templates:** Spring automatically provides the CSRF token as a hidden field in forms:

html

Copy code

<form action="/submit" method="post">

<input type="hidden" th:name="${\_csrf.parameterName}" th:value="${\_csrf.token}" />

<button type="submit">Submit</button>

</form>

1. **In HTML/JavaScript (non-Thymeleaf):** If you’re building forms manually, you can retrieve the token as follows:

java

Copy code

@GetMapping("/form")

public String form(Model model, CsrfToken csrfToken) {

model.addAttribute("csrfToken", csrfToken);

return "formPage";

}

And include it in the form:

html

Copy code

<input type="hidden" name="${csrfToken.parameterName}" value="${csrfToken.token}" />

**CSRF with REST APIs**

If you’re building a REST API (e.g., with React as the frontend), CSRF tokens are included in headers instead of forms.

1. **Frontend (React/JavaScript):** When making an XMLHttpRequest or fetch request, include the CSRF token as a header:

javascript

Copy code

fetch('/api/resource', {

method: 'POST',

headers: {

'X-CSRF-TOKEN': csrfToken,

'Content-Type': 'application/json'

},

body: JSON.stringify(data)

});

1. **Backend:** Spring Security validates the token automatically when CSRF protection is enabled.

**Testing CSRF Protection**

If CSRF protection is working, unauthorized requests without a valid CSRF token will result in a **403 Forbidden** response.

**When to Disable CSRF Protection**

CSRF protection is not always necessary, such as:

1. **Stateless APIs:** If your API does not use cookies for authentication and relies on mechanisms like OAuth 2.0, JWTs, or API keys.
2. **GET-only Endpoints:** Since CSRF only applies to state-changing requests (POST, PUT, DELETE), read-only endpoints don’t need CSRF protection.

**Conclusion**

CSRF protection in Spring Boot ensures that state-changing operations (e.g., submitting a form, making a POST request) are legitimate by requiring a token unique to the user's session. It is enabled by default in Spring Security and should only be disabled in specific use cases like stateless APIs.

Validation annotations

### In Spring Boot, validation is typically done using JSR-303/JSR-380 annotations (Bean Validation 2.0), which are part of the Java Bean Validation specification. These annotations are used to enforce rules on input data, typically for entities or DTOs. Spring Boot integrates this validation seamlessly with @Valid or @Validated annotations.

### @NotNull

* Ensures that the field is not null.

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

private String username;

### @NotEmpty

* Ensures that the field is not null and not empty (i.e., not an empty string or collection).

@NotEmpty(message = "Password cannot be empty")

private String password;

### @NotBlank

Ensures that the field is not null, not empty, and does not consist of only whitespace.

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

private String email;

### @Size

* Ensures that the field’s size is within the specified range.
* For strings, it checks the length; for collections, it checks the number of elements.
* **@Size**(min = 6, max = 12, message = "Password must be between 6 and 12 characters")private String password;

### @Min ****and**** @Max

* Ensures that the numeric value is within a given range.

@Min(18)

@Max(100)

private int age;

### @Email

* Validates that the field is a valid email address

@Email(message = "Please provide a valid email address")

private String email;

### @Pattern

* Ensures that the field matches a given regular expression.

@Pattern(regexp = "^[A-Za-z0-9]{5,10}$", message = "Username must be alphanumeric and between 5 to 10 characters")

private String username;

### @Past ****and**** @Future

* Ensures that the date field is in the past or future, respectively.

@Past(message = "Date of birth must be in the past")

private LocalDate dateOfBirth;

### @AssertTrue ****and**** @AssertFalse

* Ensures that the field is true or false.

@AssertTrue(message = "Must accept the terms and conditions")

private boolean acceptedTerms;

### @Positive ****and**** @PositiveOrZero

* Ensures that the value is a positive number or zero.

@Positive(message = "Price must be positive")

private BigDecimal price;

### @Negative ****and**** @NegativeOrZero

* Ensures that the value is a negative number or zero.

@Negative(message = "The value must be negative")

private BigDecimal balance;

### @Digits

* Ensures that the field has a specific number of integer and fraction digits.

@Digits(integer = 3, fraction = 2, message = "Amount must be a valid number with up to 3 integer and 2 fraction digits")

private BigDecimal amount;

### @Valid

* Used to trigger validation of nested objects.
* This is not a constraint but tells Spring to validate child objects (e.g., in DTOs).

@Valid private Address address;

### @NotNull ****/**** @Valid ****for Collections****

* If you want to validate a list, map, or set, you can use @Valid to trigger validation for each item in the collection.

@Valid

private List<@NotNull String> items;

### @FutureOrPresent

* Ensures that the date is in the future or present (not past).

@FutureOrPresent(message = "The event date must be in the present or future")

private LocalDate eventDate;

To apply these annotations, you would typically use the **@Valid** or **@Validated** annotation in your controllers, service, or anywhere Spring handles validation.

import javax.validation.Valid;

@PostMapping("/register")

public ResponseEntity<String> registerUser(@Valid @RequestBody UserDTO userDTO) {

// The userDTO will be validated before entering the method

userService.saveUser(userDTO);

return ResponseEntity.ok("User registered successfully!");

}

* **@Valid** triggers validation on the **DTO** object.
* If any validation fails, Spring automatically returns a 400 Bad Request response with validation error details (you can customize the error response if needed).

 **Use @NotNull, @NotEmpty, @NotBlank** for fields that require a non-null value.

 **Use @Size, @Min, @Max** to constrain the size of a string or the range of a number.

 **Use @Email, @Pattern** for pattern-based validation (e.g., for emails or specific formats).

 **Use @Valid** to validate nested objects and collections.

 **Use @AssertTrue, @AssertFalse** for boolean validations.

**Spring Boot Actuator**

To add spring actuators we need to add dependencies that as spring-boot-actuator  
**Key Features of Spring Boot Actuator**

1. **Pre-Built Endpoints**: Provides endpoints for monitoring and managing your application, such as /actuator/health and /actuator/metrics.
2. **Health Checks**: Offers a /health endpoint to report the application's health status, with built-in checks for database connectivity, disk space, etc.
3. **Metrics**: Exposes a /metrics endpoint to provide performance metrics like memory usage, request counts, and response times.
4. **Customizable**: You can define custom endpoints, health checks, and metrics to suit your application.
5. **Integration**: Easily integrates with monitoring tools like Prometheus, Grafana, or ELK (Elasticsearch, Logstash, and Kibana).

**Actuators dependencies in pom.xml**<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

### ****Common Actuator Endpoints****

| **Endpoint** | **Description** |
| --- | --- |
| /actuator | Lists all available Actuator endpoints. |
| /actuator/health | Shows the health status of the application (e.g., UP, DOWN). |
| /actuator/metrics | Displays various application metrics (e.g., JVM memory, request counts, response times). |
| /actuator/info | Displays application-specific information (e.g., version, description). |
| /actuator/env | Exposes environment properties (e.g., system properties, environment variables). |
| /actuator/beans | Provides details about all Spring beans in the application context. |
| /actuator/mappings | Shows all request mappings in the application. |

**Customizing Actuator**

**Enable/Disable Endpoints**

By default, some endpoints are disabled. You can enable them in application.properties or application.yml.

properties

management.endpoints.web.exposure.include=health,metrics,info,env

* include: Specifies which endpoints to expose.
* exclude: Specifies which endpoints to hide.

**Securing Endpoints**

It's important to secure sensitive Actuator endpoints, especially in production.

@Configuration

@EnableWebSecurity

public class SecurityConfig extends WebSecurityConfigurerAdapter {

@Override

protected void configure(HttpSecurity http) throws Exception {

http.authorizeRequests()

.requestMatchers(EndpointRequest.to("health", "info")).permitAll()

.requestMatchers(EndpointRequest.toAnyEndpoint()).hasRole("ADMIN")

.and()

.httpBasic()

} }

BCryptPassword encoder

Encrypting password with BCryptPassword encoder

in signup   
before storing convert the row data or raw password to the encrpypted password  
  
user.setPassword(passwordEncoder.encode(user.getPassword()));

userRepository.save(user);

in login

before validating have to convert again the raw data given by the user to encrypted password and then compare the recently encrypted password with the inserted encrypted password

**if** (user != **null** && passwordEncoder.matches(password, user.getPassword()))

in resetting password after checking for the username have to take the new password confirming it and the storing it before storing it have to encrypt it using BCryptPassword encoder

user.setPassword(passwordEncoder.encode(user.getPassword()));

userRepository.save(user);

### ****Core Concepts****

1. **Spring Boot Basics**
   * Introduction to Spring Boot
   * Features and advantages of Spring Boot
   * Spring Boot Starter dependencies
2. **Spring Boot Project Setup**
   * Creating a Spring Boot project using Spring Initializr
   * Understanding the application.properties and application.yml files
   * Spring Boot DevTools for development
3. **Dependency Injection**
   * @Component, @Service, @Repository annotations
   * @Autowired and @Qualifier
   * Bean lifecycle and scopes
4. **Spring Boot Configuration**
   * Externalized configuration
   * Profiles (@Profile annotation)
   * Configuration properties with @ConfigurationProperties

### ****Web Development****

1. **Building REST APIs**
   * @RestController and @RequestMapping
   * @GetMapping, @PostMapping, @PutMapping, @DeleteMapping
   * Path variables and query parameters
   * Exception handling with @ControllerAdvice and @ExceptionHandler
2. **Spring Boot and Thymeleaf**
   * Building dynamic web pages with Thymeleaf
   * Passing data between the controller and views
3. **Validation**
   * Using @Valid and @Validated
   * Bean validation with @NotNull, @Size, @Pattern, etc.
   * Custom validators
4. **Session Management**
   * HttpSession and cookies
   * Spring Session for managing sessions
   * Securing sessions

### ****Database and Persistence****

1. **Spring Data JPA**
   * Setting up Spring Data JPA with MySQL/PostgreSQL
   * Using JpaRepository and CrudRepository
   * Custom queries with JPQL and native SQL
   * Pagination and sorting
2. **Database Migrations**
   * Using Flyway or Liquibase for schema migrations
3. **Transactions**
   * Understanding @Transactional
   * Propagation and isolation levels

### ****Security****

1. **Spring Security**
   * Basics of Spring Security
   * Configuring authentication and authorization
   * Role-based access control
   * Using JWT (JSON Web Token) for authentication
2. **Cross-Origin Resource Sharing (CORS)**
   * Configuring CORS in Spring Boot

### ****Advanced Topics****

1. **Spring Boot Actuator**
   * Monitoring and managing applications with Actuator
   * Exposing metrics, health checks, and other endpoints
2. **Logging**
   * Configuring logging with Logback and SLF4J
   * Using different logging levels
3. **Spring Boot and Messaging**
   * Integrating RabbitMQ or Kafka
   * Sending and receiving messages
4. **Spring Boot and Scheduling**
   * Scheduling tasks with @Scheduled
   * Asynchronous execution with @Async
5. **Microservices with Spring Boot**
   * Building microservices using Spring Boot
   * Service discovery with Spring Cloud Netflix Eureka
   * API Gateway with Spring Cloud Gateway

### ****Testing****

1. **Testing in Spring Boot**
   * Writing unit tests with JUnit and Mockito
   * Testing REST controllers with MockMvc
   * Integration testing
2. **TestContainers**
   * Using TestContainers for database testing

Bottom of Form