**Objectives**

* Demonstrate implementation of RESTful Web Service using POST/PUT/DELETE method with input validation
  + HTTP method types (GET, POST, PUT, DELETE), REST service URL naming guidelines, @RequestMapping, @GetMapping, @PostMapping, @PutMapping, @DeleteMapping, setting POST request payload and invoking the REST service in Postman and curl, JSON to bean mapping, @RequestBody, validating input request using javax.validation and hibernate validators, @Size, @NotNull, @NotBlank, @Min, @Max, @JsonFormat, @Valid, global exception handling, handle number formatting errors
    - HTTP Request Methods - https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods
    - RESTful API naming guide - https://restfulapi.net/resource-naming/
    - Request Mapping - https://docs.spring.io/spring/docs/5.2.0.RELEASE/spring-framework-reference/web.html#mvc-ann-requestmapping
    - Validation - https://www.mkyong.com/spring-boot/spring-rest-validation-example/

**1. HTTP Request Methods**

RESTful Web Services leverage HTTP methods to perform CRUD (Create, Read, Update, Delete) operations. Each method serves a specific purpose:

|  |  |  |  |
| --- | --- | --- | --- |
| HTTP Method | Type | Purpose | Example Endpoint |
| GET | Read | Retrieve a resource | /employees |
| POST | Create | Add a new resource | /employees |
| PUT | Update | Update an existing resource | /employees/{id} |
| DELETE | Delete | Remove a resource | /employees/{id} |

**2. RESTful URL Naming Guidelines**

RESTful services should use resource-oriented URIs rather than action-based URIs. These URLs are intuitive and reflect resource hierarchy.

|  |  |  |
| --- | --- | --- |
| Action | RESTful URL | HTTP Method |
| List All | /employees | GET |
| Get by ID | /employees/{id} | GET |
| Create | /employees | POST |
| Update | /employees/{id} | PUT |
| Delete | /employees/{id} | DELETE |

**3. Spring MVC Annotations for REST Mapping**

Spring Boot provides convenient annotations for RESTful mappings:

|  |  |
| --- | --- |
| Annotation | Description |
| @RestController | Marks the class as a RESTful controller |
| @RequestMapping | Maps the base URL for the controller |
| @GetMapping | Maps HTTP GET requests |
| @PostMapping | Maps HTTP POST requests |
| @PutMapping | Maps HTTP PUT requests |
| @DeleteMapping | Maps HTTP DELETE requests |
| @RequestBody | Maps JSON request body to Java object |
| @PathVariable | Extracts variables from the URI |
| @Valid | Triggers validation on the incoming object |

**4. JSON to Java Bean Mapping with @RequestBody**

When a client sends JSON in a request body, Spring automatically maps it to a Java class using @RequestBody.

**Example:**

**JSON Payload:**

{

"id": 101,

"name": "Alice",

"department": "HR"

}

**Java Model:**

public class Employee {

private int id;

private String name;

private String department;

// Getters and setters

}

**Controller Method:**

@PostMapping("/employees")

public ResponseEntity<Employee> createEmployee(@RequestBody Employee employee) {

return new ResponseEntity<>(employeeService.save(employee), HttpStatus.CREATED);

}

**5. Input Validation Using JSR 380 (Bean Validation)**

To ensure data integrity, Spring Boot supports validation using javax.validation and Hibernate Validator.

|  |  |
| --- | --- |
| Annotation | Description |
| @NotNull | Field must not be null |
| @NotBlank | Field must not be blank (for Strings) |
| @Size | Specifies min/max length for strings or collections |
| @Min | Specifies minimum numeric value |
| @Max | Specifies maximum numeric value |
| @JsonFormat | Defines JSON serialization format (e.g: for dates) |

**Java Model with Validation:**

import javax.validation.constraints.\*;

public class Employee {

@Min(1)

private int id;

@NotBlank(message = "Name is mandatory")

private String name;

@NotBlank

private String department;

@Min(10000)

@Max(1000000)

private double salary;

// Getters and setters

}

Controller Method with @Valid:

@PostMapping("/employees")

public ResponseEntity<Employee> addEmployee(@Valid @RequestBody Employee employee) {

return ResponseEntity.ok(employeeService.save(employee));

}

**6. Global Exception Handling**

To handle errors (e.g: validation failure, number format), create a global exception handler using @ControllerAdvice.

@ControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(MethodArgumentNotValidException.class)

public ResponseEntity<String> handleValidationErrors(MethodArgumentNotValidException ex) {

return new ResponseEntity<>("Invalid input: " + ex.getMessage(), HttpStatus.BAD\_REQUEST);

}

@ExceptionHandler(NumberFormatException.class)

public ResponseEntity<String> handleNumberFormat(NumberFormatException ex) {

return new ResponseEntity<>("Number format error: " + ex.getMessage(), HttpStatus.BAD\_REQUEST);

}

}

**7. Testing with Postman and cURL**

**Postman**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Operation | Method | URL | Body Content-Type | Example Payload |
| Create | POST | http://localhost:8080/employees | application/json | {"id":101,"name":"Alice","department":"HR","salary":50000} |
| Update | PUT | http://localhost:8080/employees/101 | application/json | {"id":101,"name":"Alice B","department":"HR","salary":60000} |
| Delete | DELETE | http://localhost:8080/employees/101 | None | None |

**POST Request**:

* Method: POST
* URL: http://localhost:8080/employees
* Body: Raw → JSON

{

"id": 101,

"name": "Alice",

"department": "HR",

"salary": 50000

}

**PUT Request**:

* Method: PUT
* URL: http://localhost:8080/employees/101
* Body: JSON

**DELETE Request**:

* Method: DELETE
* URL: <http://localhost:8080/employees/101>

**cURL Examples**

|  |  |
| --- | --- |
| Operation | Command |
| POST | curl -X POST http://localhost:8080/employees -H "Content-Type: application/json" -d "{\"id\":101,\"name\":\"Alice\",\"department\":\"HR\",\"salary\":50000}" |
| PUT | curl -X PUT http://localhost:8080/employees/101 -H "Content-Type: application/json" -d "{\"id\":101,\"name\":\"Alice B\",\"department\":\"HR\",\"salary\":60000}" |
| DELETE | curl -X DELETE http://localhost:8080/employees/101 |

**POST:**

**curl -X POST http://localhost:8080/employees \**

**-H "Content-Type: application/json" \**

**-d '{"id":101,"name":"Alice","department":"HR","salary":50000}'**

**PUT:**

curl -X PUT http://localhost:8080/employees/101 \

-H "Content-Type: application/json" \

-d '{"id":101,"name":"Alice B","department":"HR","salary":60000}'

**DELETE:**

curl -X DELETE <http://localhost:8080/employees/101>

Implementing RESTful services using POST, PUT, DELETE with proper input validation is essential for creating secure, robust, and maintainable APIs. Spring Boot simplifies this with declarative annotations, validation support, and seamless integration with tools like Postman and cURL.

**Significance of HTTP Method Types in RESTful Web Services**   
  
SME to explain the importance of HTTP Method Types for RESTful Web Services. 

|  |  |
| --- | --- |
| **HTTP Method** | **Usage Scenario** |
| GET | Used to get data about a resource |
| POST | Used to create a resource |
| PUT | Used to update a resource |
| DELETE | Used to delete a resource |

The method type is just a classification and does not actually have the persistence implemented. The respective application is expected to take responsibility in implementing the persistence.

**Introduction**

RESTful Web Services are built on top of the HTTP protocol and make use of its standard methods to perform operations on resources. These HTTP method types (also called verbs) define the semantic intent of a request, providing a clear and standardized way to manage data.

Each method type corresponds to a specific CRUD operation — Create, Read, Update, or Delete — making REST services intuitive, predictable, and scalable.

**Role of Method Types in REST**

Although HTTP methods do not themselves implement data persistence, they act as semantic indicators. The application logic (controller, service, DAO layers) is responsible for performing the actual database operations like insert, update, delete, or query.

**Importance of Each Method Type**

Below is a summary of each HTTP method and its typical usage in RESTful design:

|  |  |  |
| --- | --- | --- |
| HTTP Method | Type | Usage Scenario |
| GET | Read | Used to retrieve data about a resource (e.g., get all employees or get employee by ID) |
| POST | Create | Used to create a new resource (e.g., add a new employee to the database) |
| PUT | Update | Used to update an existing resource (e.g., change the name or department of an employee) |
| DELETE | Delete | Used to delete a resource (e.g., remove an employee record) |

**Real-World Example**

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Endpoint | HTTP Method | Description |
| Fetch all users | /employees | GET | Returns a list of all employees |
| Add new employee | /employees | POST | Adds a new employee record |
| Update employee | /employees/101 | PUT | Updates details of employee 101 |
| Delete employee | /employees/101 | DELETE | Deletes employee with ID 101 |

**Key Clarification**

The method type is just a classification and does not actually have the persistence implemented.

This statement highlights an important concept:

* \*\*HTTP methods describe what should happen, but they do not define how it happens.
* The actual data operations (persistence logic) are implemented in the application backend — such as a Spring Boot REST controller calling a service method that interacts with a database.

Example:

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

public ResponseEntity<Employee> updateEmployee(@PathVariable int id, @RequestBody Employee emp) {

// Actual update logic in service layer (e.g., Hibernate or JDBC update)

return ResponseEntity.ok(employeeService.update(id, emp));

}

**Conclusion**

The use of HTTP method types in RESTful Web Services brings clarity, uniformity, and predictability to web API design. While the methods themselves do not perform data operations, they serve as critical indicators of intent, guiding both developers and client applications. The responsibility of implementing persistence lies entirely with the server-side application logic.

**SME to provide explanation about the following aspects:**

* Explain how spring framework takes care of converting the request payload into country bean
* Spring parses the JSON request payload data using Jackson parser
* For each attribute in JSON, respective method name is constructed by applying initcaps and get prefix. For example, the name attribute is changed with initcaps as Name, then get is prefixed to it which results in getName, based on this the respective method is invoked using Reflection API.
* Spring creates country object and invokes the respective setter method based on JSON data.
* The it invokes the controller method passing the country object created
* Provide explanation regarding bean naming conventions

When a client sends a JSON payload in a REST API call (usually a POST or PUT request), Spring uses **Jackson** (a JSON parsing library) to deserialize the JSON into a corresponding Java object — also called a POJO (Plain Old Java Object) or a bean.

For example, a JSON like:

{

"code": "IN",

"name": "India"

}

can be mapped to a Java class like this:

public class Country {

private String code;

private String name;

public String getCode() { return code; }

public void setCode(String code) { this.code = code; }

public String getName() { return name; }

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

}

Spring Boot automatically configures Jackson via the spring-boot-starter-web dependency. When a REST controller method uses the @RequestBody annotation, Spring uses Jackson’s ObjectMapper to convert the JSON payload into a Java object:

@PostMapping("/countries")

public ResponseEntity<String> createCountry(@RequestBody Country country) {

return ResponseEntity.ok("Country received: " + country.getName());

}

**How Jackson Maps JSON to Java Bean Using Reflection**

Spring does not manually invoke each setter. Instead, Jackson uses **Java Reflection** and naming conventions to do this dynamically. Here’s how:

1. Each key in the JSON is taken, e.g., "name".
2. The first letter is capitalized → Name.
3. The prefix set is added → setName.
4. Jackson uses reflection to check if the Country class has a setName() method.
5. If found, it invokes setName("India") on the new Country object.

This happens for all fields in the JSON. The object is populated using the appropriate setter methods and then passed as an argument to the controller method.

So when Spring receives the JSON:

{ "code": "IN", "name": "India" }

It automatically:

* Instantiates a new Country object
* Calls setCode("IN") and setName("India") using reflection
* Passes the fully populated object to the method like:

createCountry(@RequestBody Country country)

**Bean Naming Conventions**

Spring and Jackson follow JavaBean naming conventions strictly to ensure correct field-to-method mapping. These are:

* Field name should be in camelCase: name, code
* Getter method should be get + capitalized field name: getName()
* Setter method should be set + capitalized field name: setName()
* For booleans: is prefix is allowed: isActive()

| **Element** | **Convention** |
| --- | --- |
| Field Name | camelCase |
| Getter Method | getFieldName() |
| Setter Method | setFieldName() |
| Boolean Getter | isFieldName() |

**End-to-End Summary**

1. JSON is received in the HTTP request body.
2. Spring uses Jackson to parse the JSON.
3. Jackson matches each JSON key to a setter method by converting the key using the rules: capitalize → prefix with set → use reflection.
4. Spring creates the Country object and calls its setter methods.
5. The controller method is finally invoked with the created and populated Country object.

This entire process is automatic and simplifies RESTful web service development in Spring Boot.

**Question for all Learners -**What needs to be done if there is another controller EmployeeController and similar validation needs to be done for Employee payload data?  
  
SME to explain the disadvantage of the above solution.  
  
This disadvantage will be overcome in the next hands on.

To apply similar validation for an Employee payload in an EmployeeController, you would need to:

1. **Annotate fields in the Employee model class using javax.validation annotations:**

@NotNull

@Size(min = 3, message = "Employee name must be at least 3 characters")

private String name;

1. **Inside EmployeeController, repeat the validation logic manually, similar to what was done for CountryController:**

ValidatorFactory factory = Validation.buildDefaultValidatorFactory();

Validator validator = factory.getValidator();

Set<ConstraintViolation<Employee>> violations = validator.validate(employee);

List<String> errors = new ArrayList<>();

for (ConstraintViolation<Employee> violation : violations) {

errors.add(violation.getMessage());

}

if (!errors.isEmpty()) {

throw new ResponseStatusException(HttpStatus.BAD\_REQUEST, errors.toString());

}

This approach must be duplicated for every controller that requires validation.

**Disadvantage of the above solution:**

This manual validation approach has several problems:

* Code duplication: Every controller requires repeating the same boilerplate logic (factory, validator, loop, exception).
* Violates DRY principle (Don't Repeat Yourself).
* Hard to maintain: If validation handling logic needs to change, you must update it in every controller.
* Not idiomatic: Spring Boot already provides a better, cleaner, and declarative way to handle validation using @Valid.

**This disadvantage will be overcome in the next hands-on, where we will:**

* Use @Valid to trigger validation automatically.
* Use BindingResult or @ExceptionHandler to centralize error handling.
* Eliminate manual ValidatorFactory code from every controller.