Module 4 - Backend Server

Spring Boot Fundamentals

GUVI Referral Program

- Benefits
  - The referrer (student) will receive a ₹4,000 Amazon voucher.
  - The referred friend will receive a ₹5,000 discount on the course fee from the website price upon enrollment.

Warm-up

Warm-up

- 8 questions, multiple choice
- Reply format: 1 / 2 / 3 / 4
- Topics: REST endpoints, URL parts, status codes

Question 1

In a Student API, which endpoint is the best fit to get the student with id 18?

1) POST /students/18
2) GET /students?id=18
3) GET /students/18
4) PUT /students

Reply format: 1 / 2 / 3 / 4

Answer 1

In a Student API, which endpoint is the best fit to get the student with id 18?

1) POST /students/18
2) GET /students?id=18
3) GET /students/18
4) PUT /students

Correct answer: 3
Why: The id is part of the resource path, so /students/18 directly identifies a single student.

Reply format: 1 / 2 / 3 / 4

Question 2

Which request uses a query parameter?

1) GET /students?active=true
2) POST /students
3) DELETE /students/7
4) GET /students/7

Reply format: 1 / 2 / 3 / 4

Answer 2

Which request uses a query parameter?

1) GET /students?active=true
2) POST /students
3) DELETE /students/7
4) GET /students/7

Correct answer: 1
Why: Query parameters appear after the question mark and usually modify behavior or filter results.

Reply format: 1 / 2 / 3 / 4

Question 3

You create a new student successfully using POST /students. What is the best success status code?

1) 204 No Content
2) 201 Created
3) 500 Internal Server Error
4) 200 OK

Reply format: 1 / 2 / 3 / 4

Answer 3

You create a new student successfully using POST /students. What is the best success status code?

1) 204 No Content
2) 201 Created
3) 500 Internal Server Error
4) 200 OK

Correct answer: 2

Why: Creating a new resource is reported with 201 Created.

Reply format: 1 / 2 / 3 / 4

Question 4

You delete a student successfully using DELETE /students/9. What is the best success status code?

1) 400 Bad Request
2) 204 No Content
3) 201 Created
4) 200 OK

Reply format: 1 / 2 / 3 / 4

Answer 4

You delete a student successfully using DELETE /students/9. What is the best success status code?

1) 400 Bad Request
2) 204 No Content
3) 201 Created
4) 200 OK

Correct answer: 2

Why: Delete often returns 204 No Content to confirm success without returning a body.

Reply format: 1 / 2 / 3 / 4

Question 5

A client calls GET /students/abc where id should be a number. What status code best fits?

1) 400 Bad Request
2) 204 No Content
3) 404 Not Found
4) 201 Created

Reply format: 1 / 2 / 3 / 4

Answer 5

A client calls GET /students/abc where id should be a number. What status code best fits?

1) 400 Bad Request
2) 204 No Content
3) 404 Not Found
4) 201 Created

Correct answer: 1
Why: The request format is wrong because the id cannot be converted to the required type.

Reply format: 1 / 2 / 3 / 4

Question 6

A client sends invalid JSON in POST /students. What status code best fits?

1) 404 Not Found
2) 204 No Content
3) 400 Bad Request
4) 500 Internal Server Error

Reply format: 1 / 2 / 3 / 4

Answer 6

A client sends invalid JSON in POST /students. What status code best fits?

1) 404 Not Found
2) 204 No Content
3) 400 Bad Request
4) 500 Internal Server Error

Correct answer: 3

Why: The server cannot parse the request body, so the client must fix the request.

Reply format: 1 / 2 / 3 / 4

Question 7

You call GET /students/999 but student 999 does not exist. What status code best fits?

1) 500 Internal Server Error
2) 201 Created
3) 404 Not Found
4) 400 Bad Request

Reply format: 1 / 2 / 3 / 4

Answer 7

You call GET /students/999 but student 999 does not exist. What status code best fits?

1) 500 Internal Server Error
2) 201 Created
3) 404 Not Found
4) 400 Bad Request

Correct answer: 3
Why: The request is valid, but the resource does not exist.

Reply format: 1 / 2 / 3 / 4

Question 8

Why is POST /students/{id} a bad idea for creating a student?

1) POST is only for updates
2) The server should assign the id during creation, not the client
3) /students/{id} can only be used with DELETE
4) POST cannot have a body

Reply format: 1 / 2 / 3 / 4

Answer 8

Why is POST /students/{id} a bad idea for creating a student?

1) POST is only for updates
2) The server should assign the id during creation, not the client
3) /students/{id} can only be used with DELETE
4) POST cannot have a body

Correct answer: 2

Why: Creating means the resource does not exist yet. The server generates the identity and returns it.

Reply format: 1 / 2 / 3 / 4

Today's goal and the mental model

What we will build today

- We will convert the current Student project into a web API.
- We will implement endpoints that accept input from:
  - the URL path
  - the query string
  - the request body
- We will make error cases predictable:
  - the client gets the correct status code
  - the client gets a clear message that explains what to fix

Request lifecycle inside Spring MVC

- **Spring MVC** is Spring's web layer for building HTTP APIs and web apps
    - Provides the rules & plumbing for routing requests (method + path) to the right controller method
- Lifecycle
    - Step 1: request reaches the controller method that matches the path and method.
    - Step 2: Spring extracts input into Java values.
        - path variables come from the URL path
        - query parameters come from the query string
        - request body comes from JSON
    - Step 3: Spring converts types when needed.
        - string to int. Example:
            - Request: GET /students/18 *and* Controller: getStudent(@PathVariable int id
                - Spring reads "18" from the URL, converts it to the integer 18, then calls your method with id = 18.
        - JSON to Java object
    - Step 4: validation can run before service logic.
    - Step 5: service runs business logic.
    - Step 6: errors become HTTP responses.

Why do predictable failures matter?

- Clients (eg: a mobile app) build code around the API behavior.
  - status codes decide retries, UI messages, and next actions
- Incorrect status codes waste debugging time.
  - 500 suggests server failure even when the client sent bad input
- Consistent error behavior is part of the API contract.
  - success responses are not enough
  - failures must be understandable and fixable

Input sources and binding annotations

Where can input come from in an HTTP request?

- **URL path**
  - Purpose: identifies which resource you are targeting
  - Example: /students/18
- **Query string**
  - Purpose: modifies behavior, filters results, or passes optional values
  - Example: /students?active=true
- **Request body**
  - Purpose: structured data for create or update
  - Example: POST /students with JSON

Path variables

- A path variable is part of the URL path that identifies a resource.
- It answers: which specific student is this request about?
- Example requests
  - GET /students/18
  - DELETE /students/18
- How the HTTP request looks

```
GET /students/18 HTTP/1.1
Host: localhost:8080
```

- Common error case
  - GET /students/abc when id is expected to be a number
  - Spring cannot convert abc to an integer, so the request should fail as a client error

```
@GetMapping("/students/{id}")
public String getStudentById(@PathVariable int id) {
    return "Requested student id = " + id;
}
```

Query parameters

- A query parameter is part of the URL after the question mark.
- It answers: how should the server process this request?
  - filters, flags, optional knobs
- Example requests
  - GET /students?active=true
  - GET /students?sort=name&active=true
- How the HTTP request looks

```
GET /students?active=true HTTP/1.1
Host: localhost:8080
```

- Key idea
  - query parameters usually do not identify the resource
  - they modify how the server responds

```
@GetMapping("/students")
public String listStudents(@RequestParam boolean active) {
    return "Listing students, active = " + active;
}
```

Request body

- The request body carries the main data for create or update.
- Most APIs use JSON in the request body.
- Spring converts JSON into a Java object.
  - This is why the JSON must be valid and match the expected shape

```
POST /students HTTP/1.1
Host: localhost:8080
Content-Type: application/json

{
  "name": "Arun",
  "email": "arun@example.com"
}

@PostMapping("/students")
public String createStudent(@RequestBody Map<String, Object> body) {
    return "Received body keys = " + body.keySet();
}
```

Quick check

- For each request, identify from where the input comes - path, query or body parameters
- **Request A**: GET /students/7
  - Answer: 7 comes from the URL path, so it is a path variable
- **Request B**: GET /students?active=true
  - Answer: active comes from the query string, so it is a query parameter
- **Request C**: POST /students with JSON body
  - Answer: name and email come from the request body

DTOs: Shaping Requests & Responses

What is a Request DTO?

- A DTO, **Data Transfer Object**, is a plain Java class used to carry data between systems.
- A Request DTO represents the request body (eg: JSON) the client sends to your API.
- A Request DTO is used at the API boundary so the server can control:
  - which fields are allowed from the client
  - how the input is validated
- *A Request DTO is different from the model*.
  - **Model**: internal representation used by service and repository. Eg: Student class
  - **Request DTO**: external input shape used by controller. Eg: CreateStudentRequest class

Why do we use Request DTOs?

- The request body is untrusted input from outside the application.
- A Request DTO helps us design a clean API contract:
  - client sends only what the API expects
  - server decides server-owned fields like id
- It reduces accidental coupling between API input and internal model.
  - the model can evolve without forcing clients to change
- It makes validation predictable.
  - validation rules live next to the input fields they apply to
- Example: create a student
  - Client sends: name, email
  - Client does not send: id

Example: Request DTO for creating a student

```
POST /students HTTP/1.1
Host: localhost:8080
Content-Type: application/json

{
  "name": "Suraj",
  "email": "suraj@example.com"
}
```

```java
public class CreateStudentRequest {
    private String name;
    private String email;

    public CreateStudentRequest() {}

    public CreateStudentRequest(String name, String email) {
        this.name = name;
        this.email = email;
    }

    public String getName() { return name; }
    public String getEmail() { return email; }

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

@PostMapping("/students")
public String createStudent(@RequestBody CreateStudentRequest request) {
    return "Creating student with name = " + request.getName();
}
```

Validation

What is validation?

- Validation is checking if input data follows rules before processing it.
- Validation rejects bad input at the API boundary.
  - the service should not run if the input is invalid
- Common validation questions
  - Is *name* present and not blank?
  - Does *email* look like a valid email address?
- Validation exists to keep failures predictable.
  - the client gets a clear reason to fix the request

How validation works in Spring Boot

- Spring Boot can validate Request DTOs using *Bean Validation* annotations.
- Rules are written on the Request DTO fields.
- The controller enables validation using @Valid.
  - @Valid tells Spring: validate this request body before calling the controller method
- If validation fails:
  - Spring stops the request before service logic
  - Spring throws an exception that we can handle globally

```
@PostMapping("/students")
public String createStudent(@Valid @RequestBody CreateStudentRequest request) {
    return "Creating student with name = " + request.getName();
}
```

Validation annotations

- @NotBlank
  - rejects null, empty, and whitespace-only strings
  - good for name and email presence
- @Email
  - rejects strings that do not look like an email address
  - good for email format

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

public class CreateStudentRequest {

    @NotBlank(message = "name is required")
    private String name;

    @NotBlank(message = "email is required")
    @Email(message = "email must be a valid email address")
    private String email;

    public CreateStudentRequest() {}

    public CreateStudentRequest(String name, String email) {
        this.name = name;
        this.email = email;
    }

    public String getName() { return name; }
    public String getEmail() { return email; }

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

Quick check

Question: Which inputs should fail validation?

A

```
{ "name": "Rohit", "email":
"rohit@example.com" }
```

B

```
{ "name": " ", "email":
"rohit@example.com" }
```

C

```
{ "name": "Rohit", "email":
"rohit.com" }
```

Answer

- B fails because name is blank.
- C fails because email format is invalid.

Exceptions and global exception handling

What an exception means in an API?

- An exception is a signal that the normal flow cannot continue.
- In an API, exceptions commonly happen when:
  - input could not be converted to the required type
  - validation failed
  - a resource was not found
  - a business rule was violated
- Exceptions are normal in real systems.
- What matters is converting exceptions into correct HTTP responses.

Why try/catch in every controller is a bad idea

- Duplicates the same error-handling code across endpoints.
- Creates inconsistent status codes and messages.
- Becomes harder to maintain as the API grows.

```java
// StudentController.java

@GetMapping("/students/{id}")
public ResponseEntity<Student> getStudent(@PathVariable int id) {
    try {
        Student s = studentService.getById(id);
        return ResponseEntity.ok(s);
    } catch (StudentNotFoundException e) {
        return ResponseEntity.status(404).body("student not found");
    } catch (Exception e) {
        return ResponseEntity.status(500).body("server error");
    }
}
```

Global exception handling

- Global exception handling means:
  - one place to convert exceptions into HTTP responses
  - controllers focus on happy-path logic
- Spring supports this using **@RestControllerAdvice.**
  - a class marked with @RestControllerAdvice runs across all controllers
  - it can *intercept* exceptions thrown while handling requests
- Each handler method uses **@ExceptionHandler**
  - it says: if this exception happens during a request, handle it here and return a response

```
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(StudentNotFoundException.class)
    public ResponseEntity<?> handleNotFound(StudentNotFoundException ex) {
        // build response and return 404
        return ResponseEntity.status(404).body("student not found");
    }
}
```

Quick check

- Let’s say an exception was thrown in your Spring Boot application. Choose the best status code for each scenario.
- **Scenario A**: GET /students/abc where id must be a number
  - Answer: 400
- **Scenario B**: POST /students where name is blank
  - Answer: 400
- **Scenario C**: GET /students/999 where student does not exist
  - Answer: 404
- **Scenario D**: POST /students where email already exists
  - Answer: 409

Recap: Status codes and responses

HTTP status codes

- An HTTP status code is a number in the response that describes the result of the request.
- Status codes help clients (eg: a mobile application) decide what to do next.
    - show a user-friendly message
    - retry or not retry a request
    - fix the request and send again
- High-level categories
    - **2xx**: request succeeded
    - **4xx**: client sent a bad request or asked for something that does not exist
    - **5xx**: server failed while processing a valid request

Success status codes used in REST APIs

- **200 OK**
  - **Meaning**: request succeeded and the response contains a body
  - **Example**: GET /students returns a list of students
- **201 Created**
  - **Meaning**: a new resource was created
  - **Example**: POST /students creates a student
- **204 No Content**
  - **Meaning**: request succeeded and there is no response body
  - **Example**: DELETE /students/9 deletes a student successfully

```
HTTP/1.1 201 Created
Content-Type: application/json

{ "id": 18, "name": "John", "email": "john@example.com" }
```

```
HTTP/1.1 204 No Content
```

Common error status codes

- **400 Bad Request**
  - The client sent invalid input
    - Invalid JSON
    - *id* cannot be converted to the expected type, like int
    - Validation failed for request body fields
- **404 Not Found**
  - The request is valid but the resource does not exist
    - Example: GET /students/999 when student 999 does not exist
- **409 Conflict**
  - The request conflicts with a business rule
    - Example: creating a student with an email that already exists
- **500 Internal Server Error**
  - Unexpected failure on the server
  - The client cannot fix error this by changing input

Quick check

- Choose the best status code.
- **Scenario A**: POST /students with invalid JSON
  - Answer: 400
- **Scenario B**: GET /students/abc where id must be a number
  - Answer: 400
- **Scenario C**: GET /students/999 where student does not exist
  - Answer: 404
- **Scenario D**: POST /students with a duplicate email
  - Answer: 409
- **Scenario E**: DELETE /students/9 succeeds
  - Answer: 204

Live coding plan

Live coding

- Goal: convert the current project from a command-line demo into a web API.
- What will be added
  - controller layer to accept HTTP requests
  - request DTO for POST input
  - validation on request body
  - global exception handler to return correct status codes

Endpoints to build

- List students
  - GET /students
- Get a student by id
  - GET /students/{id}
- Create a student
  - POST /students
- How will the HTTP requests look?

```
GET /students HTTP/1.1
Host: localhost:8080

GET /students/18 HTTP/1.1
Host: localhost:8080

POST /students HTTP/1.1
Host: localhost:8080
Content-Type: application/json

{
  "name": "Sruthi",
  "email": "sruthi@example.com"
}
```

How will the code will be wired?

- Controller
  - Receives the HTTP request
  - Reads input using @PathVariable and @RequestBody
  - Calls the service
- Service
  - Holds business logic
  - Throws meaningful exceptions for error cases
- Repository
  - Stores and retrieves students (in-memory in this project)
- Global exception handler
  - Converts exceptions into HTTP responses
  - Prevents repeating try/catch in every controller method

Coding activity

Coding activity

- Work in groups and submit code to the GitHub Discussions thread.
- When writing the code,
  - Do not add try/catch in controller methods for these tasks
  - Throw meaningful exceptions from service
  - Handle those exceptions in the global exception handler
  - Test your endpoints using curl or Postman before submitting (please ask if you need help)

Group A task

- **Feature request**: update and delete students through the API.
- **Deliverables**
  - Update student
    - PUT /students/{id}
    - status codes
      - 200 on success
      - 404 if id does not exist
  - Delete student
    - DELETE /students/{id}
    - status codes
      - 204 on success
      - 404 if id does not exist

```
PUT /students/18 HTTP/1.1
Host: localhost:8080
Content-Type: application/json

{
  "name": "Ananya Menon",
  "email": "ananya@example.com"
}

DELETE /students/18 HTTP/1.1
Host: localhost:8080
```

Group B task

- **Feature request**: prevent duplicate emails.
- Deliverables
  - When creating a student, if the email already exists
    - throw a custom exception from the service
    - return status code 409
  - Add handling for that exception in the global exception handler.

```
POST /students HTTP/1.1
Host: localhost:8080
Content-Type: application/json

{
  "name": "Rohit",
  "email": "rohit@example.com"
}
```

Expected outcome

- status code: 409

- message explains the conflict in plain language

Submission checklist

- Before submitting, confirm:
  - Endpoints work on the happy path.
  - Error cases return correct status codes.
    - 400 for invalid input
    - 404 for missing student
    - 409 for duplicate email
  - Controller methods stay focused on request handling.
    - No repeated try/catch blocks
  - Global exception handler contains the exception-to-response mapping.

That’s a wrap!

Key takeaways

- Input binding in Spring MVC
  - @PathVariable reads from the URL path
  - @RequestParam reads from the query string
  - @RequestBody reads JSON from the request body
- Request DTO
  - defines the allowed request body shape
  - separates API input from internal model
- Validation
  - rejects bad input before service logic runs
- Global exception handling
  - central place to convert exceptions into correct HTTP responses
- Status codes are part of the API contract.