CADT IDT

W3 PRACTICE

Express Basics + POST + Middleware

At the end of this practice, you can

✓ **Create** and run a express.js HTTP server ✓ **Implement** route handling using express.js ✓ Parse form data from POST requests with middleware. ✓ Apply middleware concept to logging

Get ready before this practice!

- ✓ **Read** the following documents to understand the nature of Express.js: https://expressjs.com/
- ✓ **Read** the following documents to know more about Express.js's built-in middleware's: https://expressjs.com/en/resources/middleware.html
- ✓ **Read** the following documents to understand MDN: HTTP POST: https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Methods/POST
- ✓ **Read** the following documents to array filter: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter

How to submit this practice?

✓ Once finished, push your **code to GITHUB** ✓ Join the **URL of your GITHUB** repository on LMS

*EXERCISE 1 – **Refactoring***

Goal

✓ Take advantage of Express.js framework's flexibility and minimalism ✓
Refactor code from node.js's built-in HTTP Module

Refactor the source code of EXERCISE 2 & 3 in Week 2 to Express.js

Q1 – What challenges did you face when using the native `http` module that Express.js helped you solve?

- **Verbose code**: Native HTTP requires lots of `if/else` or `switch` statements for routing.
- **Manual response header**s: You need to explicitly set response header like `Content-Type`.
- **No build-in body parsing**: Parsing from data manually is tedious and error-prone.
- **No middleware support**: Native module doesn't support middleware; everything is done in one big callback.
- Error handling is manual and repetitive.

=> **Express.js** abstract all of these and makes server logic clean and modular.

Q2 – How does Express simplify route handling compared to the native HTTP server?

- With express, route handlers like `app.get('/about', ...)` declarative and readable.
- You don't need to manually check the HTTP method or URL.
- Express routes are easy to group and organize using routers.

Q3 – What does middleware mean in Express, and how would you replicate similar behavior using the native module?

- Middleware in Express is a function that has access `req`, `res`, and `next()`, used to perform actions like logging, parsing, authentication, etc.

EXERCISE 2 – API for Filtering and Fetching Course Records

Goal

✓ Understand Route Parameters (`:param`) ✓
Work with Query Parameters (`?key=value`) ✓
Implement Conditional Logic for Filtering ✓
Build Real-World Web API Behavior ✓
Practice Defensive Programming

For this exercise you will start with a **START CODE (EX-2)**

Context - You are building a backend API for a university's course catalog. Each course has the following fields

```
{
  "id": "CSE101",
  "title": "Introduction to Computer Science",
  "department": "CS",
  "level": "undergraduate",
  "credits": 3,
  "instructor": "Dr. KimAng",
  "semester": "fall"
}
```

Task:

1. **Create a route:**
   `GET /departments/:dept/courses`

   Example:
   `/departments/CSE/courses`

2. **Accept query parameters** to **filter** the result:
   - `level` → e.g., `undergraduate`, `graduate`
   - `minCredits` → integer
   - `maxCredits` → integer
   - `semester` → `fall`, `spring`, etc.
   - `instructor` → partial match

   Example:
   `/departments/CSE/courses?level=undergraduate&minCredits=2&semester=fall`

3. **Return** a JSON array of courses that match:
   - The `:dept` from the route parameter

    - The filter criteria from query parameters
4. **Edge Cases to Handle:**
    - Invalid credit ranges (`minCredits > maxCredits`)
    - No matching courses
    - Missing or unsupported query parameters (ignore them silently)

Example Response:

Request:
`/departments/CSE/courses?level=undergraduate&minCredits=3&instructor=KimAng`

Response:

```
json CopyEdit
{
  "results": [
    {
      "id": "CSE101",
      "title": "Introduction to Data Science",
      "department": "CSE",
      "level": "undergraduate",
      "credits": 3,
      "instructor": "Dr. KimAng",
      "semester": "fall"
    }
  ],
  "meta": {
    "total": 1
  }
}
```

Test Example URLs:

- `http://localhost:3000/departments/CSE/courses`
- `http://localhost:3000/departments/CSE/courses?level=undergraduate`
- `http://localhost:3000/departments/CSE/courses?minCredits=4`
- `http://localhost:3000/departments/CSE/courses?instructor=smith&semester=fall`

EXERCISE 3 – Implement and Apply Middleware to Enhance Your API

Goal

Your goal is to modularize and secure your course filtering API using **Express middleware**. Middleware helps keep your code clean, reusable, and extensible.

TASK 1 - Create a middleware function that logs the following for every request:

- HTTP method (GET, POST, etc.)
- Request path (e.g., `/departments/CSE/courses`)
- Query parameters
- Timestamp in ISO format

Apply this middleware globally so it logs **all incoming requests** to the server.

TASK 2 - Create a route-specific middleware to validate query parameters:

- If `minCredits` or `maxCredits` are present, ensure they are valid integers.
- If `minCredits > maxCredits`, return `400 Bad Request` with an error message.

Apply this middleware only to the `/departments/:dept/courses` route.

TASK 3 – (Optional Bonus) Token-Based Authentication Middleware

Simulate basic API security:

- Require a `token` query parameter (e.g., `?token=xyz123`)
- If the token is missing or incorrect, respond with `401 Unauthorized`.

This middleware can be applied either globally or to specific routes.

Deliverables

- `logger.js` – contains your logging middleware.
- `validateQuery.js` – contains your validation middleware.
- `auth.js` (optional) – contains your token authentication middleware.
- `server.js` – where you apply middleware and define the course filtering route.

Test Cases to Try

- `GET /departments/CSE/courses?minCredits=abc` → should return `400 Bad Request`
- `GET /departments/CSE/courses?minCredits=4&maxCredits=2` → should return `400 Bad Request`

- `GET /departments/CSE/courses?token=xyz123` → should succeed if token middleware is active

Reflective Questions (Submit in separate PDF file)

Middleware & Architecture

1. What are the advantages of using middleware in an Express application?
   Middleware in Express offers several advantages:
   - **Separation of concerns** – Isolates tasks like logging, validation, and authentication.
   - **Reusability** – Can be reused across multiple routes and projects.
   - **Readability** – Makes route handlers cleaner and easier to understand.
   - **Scalability** – Easy to add new features or modify behavior without rewriting routes.

2. How does separating middleware into dedicated files improve the maintainability of your code?
   It improves maintainability by:

   - **Reducing clutter** in your main server file (`server.js`).
   - **Encouraging modular design** so individual middlewares can be updated independently.
   - **Improving collaboration**, allowing different developers to work on different middlewares simultaneously.

3. If you had to scale this API to support user roles (e.g., admin vs student), how would you modify the middleware structure?

   I would add a **role-based authorization middleware** to:

   - Decode user information from tokens (e.g., JWTs).
   - Attach user roles (admin, student) to the request.
   - Protect sensitive routes (e.g., course creation, deletion) using route-specific role checks.

Query Handling & Filtering

4. How would you handle cases where multiple query parameters conflict or are ambiguous (e. g., **`minCredits=4`** and **`maxCredits=3`**)?
   I would implement query **conflict resolution logic** in the middleware. For instance:
   - Respond with a `400 Bad Request`.
   - Provide a **clear error message** to the client.
   - Optionally, auto-correct based on business logic or prompt the user.

5. What would be a good strategy to make the course filtering more user-friendly (e.g., handling typos in query parameters like "falll" or "dr. smtih")?

   - Use **fuzzy string matching** (e.g., with `fuse.js`) to match near terms.
   - Provide **autocomplete** or **suggested corrections** in API responses.
   - Implement **fallback filters** that try partial matches or soundex-based logic.

Security & Validation

6. What are the limitations of using a query parameter for authentication (e. g., `?token=xyz123`)? What alternatives would be more secure?

   Limitations:

   - Query parameters can be **logged** in server logs and browser history.
   - They are **visible** in URL sharing and easy to manipulate.

   More secure alternatives:

   - **Bearer tokens** in the `Authorization` header.
   - Use of **JWT (JSON Web Tokens)** with proper encryption.
   - **OAuth 2.0** for delegated access in production environments.

7. Why is it important to validate and sanitize query inputs before using them in your backend logic?

   - Prevents **type errors** and invalid operations.
   - Protects against **security threats** like injection attacks.
   - Ensures that **application logic behaves as expected** without crashing.

Abstraction & Reusability

8. Can any of the middleware you wrote be reused in other projects? If so, how would you package and document it?

   Yes.

   - `logger.js`, `validateQuery.js`, and `auth.js` are all reusable.
   - I would package them as NPM modules or include them in a shared utility repo.
   - Documentation would include **usage examples**, **expected input**, and **configuration options**.

9. How could you design your route and middleware system to support future filters (e.g., course format, time slot)?

- Keep middleware **composable and modular** (one responsibility per function).
- Use a **dynamic query filter builder** to handle multiple optional parameters.
- Maintain a clear **schema of valid filters** and apply sanitization + validation for each.

Bonus – Real-World Thinking

10. How would this API behave under high traffic? What improvements would you need to make for production readiness (e.g., rate limiting, caching)?

Under high traffic, the API might:

- Experience **latency spikes** or **downtime**.
- Suffer from **rate abuse** or **DoS attacks**.

Improvements:

- Implement **rate limiting** using middleware like `express-rate-limit`.
- Use **in-memory caching** (e.g., Redis) for frequent requests.
- Deploy using **load balancers**, **CDNs**, and **horizontal scaling** for resilience.