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

Goals

✓ 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?

Using the native http module required a lot of manual work like parsing URLs, handling routes, and writing long if-else statements. Express made this easier with built-in routing, cleaner code, and easy access to request data.

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

Express lets you define routes in a simple and readable way using app.get(), app.post(), etc., instead of checking URL and method manually. It also handles route parameters and queries for you automatically.

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 runs before the final request handler. It's used for things like logging and authentication. In the native module, you'd have to write and call those functions yourself inside the main server code, which is harder to manage.

EXERCISE 2 – *API for Course Records*

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

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

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"
}
```

Q1 - Create a route

```
GET /departments/:dept/courses
```

EXAMPLE

```
/departments/CSE/courses
```

Q2 - 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
```

Q3 - Return a JSON array of courses that match:

- The :dept from the route parameter
- The filter criteria from query parameters

Q4 – Handle Edge Cases

- **Invalid credit ranges** (minCredits > maxCredits)
- No **matching courses**
- **Missing** or **unsupported** query parameters (ignore them silently)

EXAMPLES

REQUEST

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

RESPONSE

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

EDGE CASES

- `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 – Enhance an API with Middleware

Goal

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

Q1 - 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.

Q2 - 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.`

Q3 – (*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

```
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

For this part, submit it in separate PDF files

Middleware & Architecture

1. What are the advantages of using middleware in an Express application?
2. How does separating middleware into dedicated files improve the maintainability of your code?
3. If you had to scale this API to support user roles (e.g., admin vs student), how would you modify the middleware structure?

Answer

1. Advantages of Middleware in Express

- Middleware lets you separate logic like logging, validation, and authentication from your main routes.
- It keeps code clean, modular, and reusable.
- You can apply middleware globally or locally, giving flexibility.

2. Benefit of Separate Middleware Files

- Improves readability and organization.
- Makes it easier to debug, test, and reuse code.
- Supports scalability when your app grows larger.

3. Supporting User Roles (Admin vs Student)

- Create role-based middleware to check permissions.
- Example: checkRole('admin') middleware before admin routes.

Query Handling & Filtering

4. How would you handle cases where multiple query parameters conflict or are ambiguous (e.g., **minCredits=4** and **maxCredits=3**)?
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”)?

Answer

4. Handling Conflicting Query Parameters

- Example: minCredits=4 and maxCredits=3 is invalid.
- Use validation middleware to check logic and return a 400 Bad Request.
- Always validate before processing.

5. Making Filtering More User-Friendly

- Use case-insensitive and partial matching.
- Use libraries like fuse.js or custom logic to handle typos (e.g., "falll" → "fall").
- Add default values or suggestions if no match is found.

Security & Validation

6. What are the limitations of using a query parameter for authentication (e.g., **?token=xyz123**)? What alternatives would be more secure?
7. Why is it important to validate and sanitize query inputs before using them in your backend logic?

Answer

6. Limitations of Query Parameter Authentication

- Query parameters like ?token=xyz123 are exposed in URLs, logs, and browser history.
- More secure alternatives:
  - HTTP headers (Authorization: Bearer <token>)
  - Sessions with cookies
  - Use OAuth or JWT for secure auth in real apps.

7. Why Validate and Sanitize Inputs

- Prevents security vulnerabilities like SQL Injection or crashes.
- Ensures your app only processes expected and safe data.
- Good validation leads to robust and reliable APIs.

Abstraction & Reusability

8. Can any of the middleware you wrote be reused in other projects? If so, how would you package and document it?
9. How could you design your route and middleware system to support future filters (e.g., course format, time slot)?

Answer

8. Reusing Middleware

- Yes! Middleware like logger and validateQuery is reusable.
- To reuse:
    - Export it as a module (module.exports)
    - Create an npm package or place in a middleware/ folder
    - Add README docs explaining how to use it.

9. Future-Proofing Middleware

- Use generic, parameter-based filtering (e.g., loop through supported filters).
- Structure middleware to add new filters easily (e.g., course format, time).
- Keep route logic clean and flexible by using chaining and config files.

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)?

Answer

10 .Production Readiness for High Traffic

- Under high traffic, the API might slow down or crash.
- Improve it with:
    - Rate limiting (e.g., limit requests per IP)
    - Caching (e.g., store common results in memory)
    - Database optimization
    - Load balancing and clustering
    - Use tools like Redis, Nginx, and PM2 for performance and scaling.