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?

Lack of syntax and lack of understanding of the concept including flow of express.

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

Reduces code and easy to understand and program compared to native HTTP server but when using express I need to install (npm install express) while using native HTTP server don't require to do that.

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

Middleware is the step which let client request pass across the express and go to response.

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?

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")?

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?

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)?

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:

1. **Middleware advantages:**
   Easier code organization, reusable logic, and clear separation of concerns (like logging, auth, validation).
2. **Dedicated middleware files:**
   Keeps code clean, easier to find bugs, and lets teams work on different features without conflicts.
3. **Scaling for user roles:**
   Add role-checking middleware and apply it to routes that need special permissions (e.g., only admins can edit).
4. **Conflicting query params:**
   Detect conflicts (like minCredits > maxCredits) and return a clear error to the user.
5. **User-friendly filtering:**
   Use fuzzy matching, suggest corrections, or ignore minor typos to help users get results even with mistakes.
6. **Query param auth limits:**
   Tokens in URLs are insecure (can leak in logs/history). Use headers, sessions, or OAuth/JWT for better security.
7. **Validate/sanitize inputs:**
   Prevents attacks, bugs, and bad data from breaking your app or exposing vulnerabilities.
8. **Middleware reusability:**
   Yes! Export as npm packages or modules, add usage docs, and import into other projects as needed.
9. **Future filters:**
   Write flexible middleware that checks for any filter in req.query, and use modular route handlers for easy updates.
10. **High traffic/production:**
    Add rate limiting, caching, error handling, and maybe clustering/load balancing to keep the API fast and reliable.