Student's name: ***Someth Phay***

IDTB100019

SE G2 Gen10 Year2 Term3

Lect. Kim Ang Kheang

S2 - PRACTICE - ExpressJS 1

Reflection Questions

GitHub Repo:

https://github.com/PhaySometh/Y2_Term3_W3_S2-PRACTICE-ExpressJS_1.git

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?

➔ Middleware provides a way to modularize and organize logic that applies to all or specific requests. It allows for cleaner code by separating concerns such as logging, authentication, and validation. Middleware helps avoid repetition, improves scalability, and makes debugging easier by adding layers of control over request-response processing.

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

➔ Separating middleware into files like logger.js, validateQuery.js, and auth.js follows the Single Responsibility Principle. This makes the codebase easier to read, test, and update. If a bug arises in validation logic, I only need to check validateQuery.js. It also encourages reusability and makes onboarding easier for new developers.

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 create a `roleMiddleware.js` file to check the user's role from a token or session and apply permissions accordingly. Middleware could be applied route-specifically (e.g., only admins can POST new courses). A role-based access control (RBAC) structure could also be introduced to define permissions centrally and enforce them across 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**)?

➔ This is already handled in validateQuery.js by checking if minCredits > maxCredits. A 400 Bad Request is returned with an explanation. More complex conflicts could use a "conflict resolution strategy" — either prioritizing one parameter or rejecting the request with helpful guidance.

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")?

➔ To improve user-friendliness:
   - Use fuzzy matching (e.g., Fuse.js) for typos.
   - Provide suggestions or correct common typos automatically.
   - Normalize inputs (e.g., lowercase, remove whitespace).
   - Include query feedback in the response (e.g., `"Did you mean 'fall'?"`).
   - Create a fixed set of allowed terms (e.g., semesters) and match input against them.

Security & Validation

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

➔ Query parameters are visible in browser history, logs, and URLs, making them insecure for sensitive tokens. Better alternatives include:
  - **HTTP headers** (e.g., `Authorization: Bearer <token>`)
  - **Session-based authentication**
  - **JWT (JSON Web Token)** with expiry and role encoding
  - **OAuth** for third-party login integration

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

➔ Validation ensures inputs are correct, preventing crashes or logic errors. Sanitization protects against injection attacks (e.g., SQL or NoSQL injection) and ensures that malicious input can't compromise the system. It's a critical part of backend security and stability.

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. The `logger`, `auth`, and `validateQuery` middleware can be reused. I would:
  - Export each middleware in individual files.
  - Create a `middleware` folder or NPM package.
  - Add README documentation for each middleware's purpose and usage.
  - Support custom config (e.g., `validTokens`, `logLevels`, etc.).

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

➔ I would:
- Use a modular filterCourses function that accepts all query parameters as input.
- Add middleware or utilities that validate and normalize new filters.
- Keep filters dynamic by looping through allowed parameters, rather than hardcoding them.
- Group filters logically (e.g., credit-related, instructor-related) to apply validations easily.

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 could slow down or crash. To prepare for production:
  - **Add rate limiting** (e.g., with express-rate-limit) to prevent abuse.
  - **Use caching** (e.g., Redis or memory caching) to store frequent query results.
  - **Deploy with load balancing** and horizontally scalable architecture.

- **Add monitoring/logging tools** like New Relic or Prometheus.
- **Move course data to a database** instead of loading from a JS file.