CADT IDT

W3 PRACTICE

Express Basics + POST + Middleware

EXERCISE 1 – Refactoring

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

Challenges with native `http` that Express solves:

- Manual routing and URL parsing
- No request body parsing
- No built-in middleware supports
- Verbose and repetitive code

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

How Express simplifies route handling:

- Clean syntax like `app.get('/path', handler)`
- Built-in support for route and query parameters
- Grouped routes and modular structure
- Less boilerplate compared to manual `if/else` in native `http`

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

- **middleware** = functions that run before the route handler (e.g., logging, auth).
- In native `http`, you must manually call middleware-like functions and manage the flow using callbacks.

EXERCISE 3 – Enhance an API with Middleware

REFLECTIVE QUESTIONS

For this part, submit it in separate PDF files

Middleware & Architecture

1. Advantages of using middleware in Express:
   - Modular: Handles tasks like logging, validation, and authentication cleanly.
   - Reusable: Shared logic across routes.
   - Composable: Runs in order, allowing clear separation of concerns.
2. How separating middleware into files helps maintainability:
   - Clear organization.

- Easier to debug and test individual functions.
- Encourages reuse and cleaner architecture.

3. Scaling for user roles (admin vs student):
    - Create role-check middleware (`checkRole('admin')`).
    - Apply conditionally to routes based on access level.
    - Store roles in user tokens or sessions for flexibility.

Query Handling & Filtering

4. Handling conflicting/ambiguous query parameters:
    - Validate early: If `minCredits > maxCredits`, return a 400 error.
    - Provide helpful error messages to guide users.
5. Making filtering user-friendly:
    - Use fuzzy matching for instructor names (`dr. smtih` → `Dr. Smith`).
    - Implement a suggestions or fallback mechanism for typos (e.g., `falll` → `fall`).
    - Normalize inputs (lowercase, trim spaces).

Security & Validation

6. Limitations of token in query:
    - Visible in URLs (logs, browser history).
    - Easy to leak or manipulate.

    Better alternatives:
    - Use Authorization headers with JWTs.
    - Implement HTTPS, sessions, or OAuth2.
7. Importance of validating/sanitizing query inputs:
    - Prevents logic bugs and crashes.
    - Mitigates injection attacks and malformed data.
    - Ensures consistent, expected behavior.

Abstraction & Reusability

8. Reusing middleware in other projects:
    - Yes—logger, validator, and auth middleware are generic.
    - Package as NPM module or utility files with clear docs and examples.
9. Designing for future filters:
    - Use a dynamic filter function that checks `req.query`.
    - Allow config-based validation rules (e.g., schema validation with Joi or Zod).
    - Keep filtering logic separate from routing.

Bonus – Real-World Thinking

10. High-traffic considerations and production improvements:
    - **Rate Limiting**: Prevent abuse with middleware (e.g., `express-rate-limit`).
    - **Caching**: Use in-memory (Redis) or HTTP cache headers.
    - **Load Balancing**: Use multiple instances with a load balancer.
    - **Monitoring**: Add logging and error tracking (e.g., with Winston, Sentry).