CADT IDT

W4 PRACTICE

REST API Design + Modular Express

At the end of this practice, you can

- ✓ Build a RESTful API for managing Articles.
- ✓ Understand and implement separation of concerns in Express (controllers, routes, models, middleware).
- ✓ Perform CRUD operations (Create, Read, Update, Delete) using REST principles. ✓ Use dynamic route parameters (:id), query strings, and request body data.

Get ready before this practice!

- ✓ **Read** the following documents to understand Rest API Principles: https://restfulapi.net/

- ✓ **Read** the following documents to know more about MCV pattern: https://www.geeksforgeeks.org/model-view-controllermvc-architecture-for-nodeapplications/

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

- ✓ Understand and apply the separation of concerns principle in Express.js.
- ✓ Organize Express.js applications into controllers, routes, models, and middleware. ✓ Use meaningful folder structures and naming conventions for maintainability.

*For this exercise you will start with a **START CODE (EX-1)***

Context

You are provided with a simple server.js file containing all the logic in one place. Your task is to **refactor** this file by separating concerns into appropriate directories:

Tasks

1. **Understand the initial code in server.js.**
2. Create the following folders:
   - controllers/ ○ routes/ ○ models/ ○ middleware/
3. Refactor the code based on the roles of each part:
   - Move request logic to controllers/ ○ Move route definitions to routes/ ○ Move user data management to models/ ○ Add a logging middleware to middleware/
4. Ensure the server.js file only contains server setup and middleware registration.
5. Maintain consistent naming and structure as described below.

Folder Structure & Naming Convention

```
project/
│
├── controllers/
│   └── userController.js
│
├── routes/
│   └── userRoutes.js
│
├── models/
│   └── userModel.js
│
├── middleware/
│   └── logger.js
│
├── server.js
├── package.json
└── README.md
```

Folder Structure & Naming Convention

Element	Convention	Example
Controllers	`camelCase.js`	`userController.js`
Routes	`camelCase.js`	`userRoutes.js`
Models	`camelCase.js`	`userModel.js`
Middleware	`camelCase.js`	`logger.js`

Bonus Challenge (Optional)

Implement a middleware that validates if the request body contains name and email before it reaches the controller.

Reflective Questions

1. **Why is separating concerns (routes, controllers, models, middleware) important in backend development?**

   Separating concerns helps organize code by responsibility, making it easier to maintain, debug, and scale. Each part (routes, controllers, models, middleware) handles a specific task, reducing complexity and preventing code duplication.

2. **What challenges did you face when refactoring the monolithic server.js into multiple files?**

   Common challenges include managing imports/exports, ensuring correct file paths, and understanding how to properly structure middleware and route handlers. It can also be tricky to maintain consistent error handling and data flow between modules.

3. **How does moving business logic into controllers improve the readability and testability of your code?**

   Controllers isolate business logic from routing, making route files cleaner and easier to read. This separation allows you to write unit tests for controllers without needing to simulate HTTP requests, improving testability

4. **If this project were to grow to support authentication, database integration, and logging, how would this folder structure help manage that growth?**

   logging, how would this folder structure help manage that growth?
   A modular folder structure allows you to add new features (like authentication or database models) in dedicated files or folders without cluttering existing code. This makes it easier to locate, update, or debug specific functionality as the project grows.

EXERCISE 2 – *RESTful API for Articles*

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

Goals ✓ Design and implement a RESTful API that follows best practices. ✓ Perform full CRUD operations (Create, Read, Update, Delete) on an Article resource. ✓ Apply REST principles such as using appropriate HTTP methods, resource-based routing, and status codes.
✓ Structure an Express.js project in a modular, maintainable way using models, controllers, and middleware.

Context

You are a backend developer at a news company. The company needs a basic REST API to manage articles, journalists, and categories. Your job is to implement this API using Express.js with dummy JSON data (no database is needed).

API Endpoints to Implement (Keep in mind to apply separation of concern, controllers, models, routes)

1. Articles Resource

- `GET /articles` — Get all articles
- `GET /articles/:id` — Get a single article by ID
- `POST /articles` — Create a new article
- `PUT /articles/:id` — Update an existing article
- `DELETE /articles/:id` — Delete an article

2. Journalists Resource

- `GET /journalists` — Get all journalists
- `GET /journalists/:id` — Get a single journalist • `POST /journalists` — Create a new journalist
- `PUT /journalists/:id` — Update journalist info
- `DELETE /journalists/:id` — Delete a journalist
- `GET /journalists/:id/articles` — Article by specific journalist

3. Categories Resource

- `GET /categories` — Get all categories
- `GET /categories/:id` — Get a single category
- `POST /categories` — Add a new category
- `PUT /categories/:id` — Update a category

- `DELETE /categories/:id` — Delete a category
- `GET /categories/:id/articles` — Articles from a categories

Reflective Questions

1. How do sub-resource routes (e.g., `/journalists/:id/articles`) improve the organization and clarity of your API?

   Sub-resource routes clearly express relationships between resources (e.g., which articles belong to which journalist). This makes the API more intuitive, easier to navigate, and helps consumers understand data hierarchy and context.

2. What are the pros and cons of using in-memory dummy data instead of a real database during development?

   Pros:

- Fast setup, no external dependencies.
- Easy to reset and modify data for testing.
- Good for prototyping and learning.

   Cons:

- Data is lost on server restart.
- Not suitable for concurrent/multi-user scenarios.
- Doesn't scale or reflect real-world persistence and querying.

3. How would you modify the current structure if you needed to add user authentication for journalists to manage only their own articles?

- Implement authentication middleware (e.g., JWT).
- Require login for protected routes.
- In article routes, check the authenticated journalist's ID matches the journalistId on the article before allowing updates/deletes.
- Add authorization checks in controllers.

4. What challenges did you face when linking related resources (e.g., matching `journalistId` in articles), and how did you resolve them?

Challenges include ensuring referential integrity (the journalistId exists in the journalists list) and handling cases where related resources are missing. This can be resolved by validating IDs before creating or updating articles and returning appropriate errors if references are invalid.

5. If your API were connected to a front-end application, how would RESTful design help the frontend developer understand how to interact with your API?

   RESTful design uses predictable, resource-oriented URLs and standard HTTP methods, making it easier for frontend developers to understand how to fetch, create, update, or delete resources. Consistent patterns reduce confusion and speed up integration.