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/ o routes/ o models/ o middleware/
3. Refactor the code based on the roles of each part:
   - Move request logic to controllers/ o Move route definitions to routes/ o Move user data management to models/ o 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?**

Answer: Separating concerns (routes, controllers, models, middleware) is important in Backend Development because it help enhances maintainability, scalability and readability. It's also help define API endpoints, controller handle, manage data and middleware processes cross-cutting concern.

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

Answer: Challenges I faced when refactoring the monolithic server.js into multiple files are incorrect folder name led to error, middleware mis-configuration and ensuring that the import are from correct folder/file.

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

Answer: Moving business logic into controllers improve the readability and testability of my code are by the requests handling from routing and data management. It improves the readability in the server.js file and it also for testability, controllers ban be tested independently with req and res and it allow unit tests to focus on each logic of the code.

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

Answer: This folder structure help manage the growth if this project were to grow to support authentication, database integration and logging would be by adding the auth (Authentication) middleware, adding database, extending the logging to be globally, …

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?

Answer: To improve the organization and clarify of my API in the sub-resource routes are by showing relationships between resources (e.g., journalists and their articles), following RESTful conventions for intuitive and hierarchical endpoint design.

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

Answer: The pros and cons of using in-memory dummy data instead of real database during development are:
Pros: Quick setup, no external dependencies, ideal for prototyping.

Cons: Data loss on restart, poor scalability, lacks real database constraints.

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

Answer: I would modify the current structure if I needed to add user authentication for journalists to manage only their own article by adding the auth (authentication) to verify journalist identity, apply middleware to routes, ensuring journalists can only manage their own articles by comparing req.journalistId with article.journalistId.

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

Answer: The challenges I faced when linking related resource are matching journalistId in articles was tricky for consistency and I would resolve them by using filter (e.g., articles.filter(a => a.journalistId === id)), with potential future validation for journalistId existence.

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?

Answer: If my API were connected to a frontend application when RESTful design help the frontend developer understand how to interact with my API are by provides predictable endpoints (e.g., GET /articles), standard HTTP methods, and clear status codes (e.g., 201, 404), making the API more active for frontend developers to fetch, create, or filter data without extensive documentation.