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-node-applications/

How to submit this practice?

- ✓ Once finished, push your **code to GITHUB**
- ✓ Join the **URL of your GITHUB** repository on LMS

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

IDTB100019

SE G2 Gen10 Year2 Term3

Lect. Kim Ang Kheang

S2 – Rest API Design

Reflection Questions

GitHub Repo:

https://github.com/PhaySometh/Y2_Term3_W3_S2-REST_API_Design.git

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 makes the codebase easier to understand, maintain, and
   scale. Each part of the application has a clear responsibility, reducing
   complexity and making it easier to debug or update specific features without
   affecting unrelated code.
   ```

2. **What challenges did you face when refactoring the monolithic server.js into multiple files?**
   ➔
   ```
   Common challenges include managing import/export syntax, ensuring correct
   file paths, and understanding how to properly organize logic between files.
   It can also be tricky to debug issues caused by missing or incorrect exports
   and to maintain consistent naming conventions.
   ```

3. **How does moving business logic into controllers improve the readability and testability of your code?**
   ➔
   ```
   Controllers isolate business logic from routing, making each file simpler
   and more focused. This separation allows for easier unit testing of logic
   without involving HTTP request/response objects and improves overall code
   readability.
   ```

4. **If this project were to grow to support authentication, database integration, and logging, how would this folder structure help manage that growth?**
   ➔
   ```
   A modular folder structure allows new features (like authentication or
   database logic) to be added as separate modules or middleware without
   cluttering existing code. This organization makes it easier to scale,
   maintain, and collaborate on the project as it 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, making the
API more intuitive and easier to navigate. They help clients understand how
data is connected (e.g., which articles belong to which journalist) and support
RESTful principles by structuring endpoints in a logical, hierarchical way.
```

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

➔

```
*Pros:*
- Fast and easy to set up
- No external dependencies
- Great for prototyping and learning
*Cons:*
- Data is lost when the server restarts
- Not suitable for production
- Cannot handle large datasets or concurrent access
```

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

➔

```
I would add authentication middleware (e.g., JWT) to verify journalist identity
on protected routes. Controllers would check the authenticated journalist’s ID
against the `journalistId` in articles to ensure only the owner can create,
update, or delete their articles.
```

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

➔

```
Challenges included ensuring IDs matched correctly and handling cases where
related resources did not exist. I resolved them by validating IDs, checking
for existence before linking, and returning appropriate error responses if a
related resource was missing.
```

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

➔

```
RESTful design uses predictable, resource-based URLs and standard HTTP methods,
making it easier for frontend developers to understand how to fetch, create,
update, or delete data. Clear endpoint naming and consistent responses improve
developer experience and reduce confusion.
```