## Exercise 1:

**1. Why is separating concerns important?**

It keeps the code organized and easier to maintain, lets different parts be developed independently, and makes testing simpler.

**2. What challenges did you face refactoring the monolithic server.js?**

Figuring out dependencies, organizing files, managing imports/exports, and making sure the app still works without errors were the main challenges.

**3. How does moving business logic to controllers help?**

It makes routes cleaner, improves code readability, and allows testing business logic separately from the routes.

**4. How would this folder structure help with future growth like authentication or database integration?**

It makes it easier to add new features by keeping code modular and organized, so new parts like authentication or logging won't mess up existing code.

## Exercise 2:

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

Sub-resource routes improve the organization and clarity by logically grouping related data under a parent resource. For example, /journalists/:id/articles clearly indicates that the articles returned belong to a specific journalist. This structure makes the API intuitive and easy to navigate, especially when consuming data that is hierarchically related. It also follows REST principles, making it predictable for frontend developers.

2. **What are the pros and cons of using in-memory dummy data instead of a real database during development?**
   a. *Pros:*
      i. Quick setup and easy to modify.
      ii. Great for prototyping and testing API endpoints.
      iii. No need for complex database configuration or queries.
   b. *Cons:*
      i. Data is not persistent and resets every time the server restarts.
      ii. Cannot handle large datasets or concurrent access properly.
      iii. Doesn’t reflect real-world database behavior (e.g., constraints, indexing).
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 an authentication system using middleware, such as JWT (JSON Web Tokens). Each journalist would log in and receive a token. The token would be verified on protected routes. Then, I would ensure that create, update, and delete article routes check that the journalistId from the token matches the journalistId in the article before allowing access. This would enforce that journalists can only manage their own articles.

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

The main challenge was filtering articles based on foreign key references like journalistId or categoryId. To solve this, I used JavaScript's Array.prototype.filter() method to return only the articles that match the given ID from the URL parameter. This approach is simple and efficient for in-memory data.

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 provides a consistent structure using HTTP methods and predictable routes. A frontend developer can easily understand how to fetch all articles (GET /articles), create an article (POST /articles), or get a journalist's articles (GET /journalists/:id/articles) without needing extra documentation. This improves developer experience and speeds up frontend-backend integration.