Ahdus Technology

# NestJS Interview Coding Task: Task Assignment Microservice

## Scenario

|  |  |
| --- | --- |
|  | You are developing a **Task Assignment** service as part of a microservices-based project management system. This NestJS service is responsible for managing tasks and their assignment to users. In a real-world setup, **User Management** would be a separate microservice; here, we will simulate minimal user data and focus on the Task service. The service should expose a REST API for basic CRUD operations on tasks, handle assignments to users, and demonstrate NestJS best practices (modules, controllers, providers, etc.) in a **microservice-friendly design**. |

## Requirements

|  |  |
| --- | --- |
|  | Implement a NestJS application (or module) that fulfills the following: |

## Task Data Model

|  |  |
| --- | --- |
|  | Use PostgreSQL (via **Drizzle ORM**) to store tasks. Each task should have at minimum: id (UUID or number), title, description (text), an assignedUserId (reference to a user, optional), and a status (e.g., "pending" or "completed"). Define the schema using Drizzle's type-safe definitions. |

## Core Endpoints (CRUD)

|  |  |
| --- | --- |
|  | Create a NestJS **TasksController** with routes for: |

* **Create Task** – POST /tasks: Creates a new task. Accepts JSON body (e.g., { title, description, assignedUserId? }). If an assignedUserId is provided, the service should verify that the user exists (simulate a call to the User service). On success, return the created task (including its generated id).
* **Get Tasks** – GET /tasks: Retrieves a list of tasks. Support filtering by assignedUserId via query param (e.g., /tasks?assignedUserId=123) to get tasks for a specific user. If no filter, return all tasks.
* **Get Task by ID** – GET /tasks/:id: Retrieves a single task by its ID. If not found, respond with an appropriate exception (e.g., 404 Not Found).
* **Update Task** – PATCH /tasks/:id: Updates a task’s details. Accept a JSON body allowing updates to the title, description, assignedUserId (to reassign), or status (e.g., marking as completed). If reassigning, verify the new user exists. Return the updated task data.
* **Delete Task** – DELETE /tasks/:id: Deletes a task. This operation should be **restricted to admin users only**.

## Microservice Interaction (Simulation)

|  |  |
| --- | --- |
|  | For this exercise, instead of actually calling a separate user microservice, simulate the **UserService**: |

* Create a simple **UsersModule/UsersService** that the Tasks service can use to validate or fetch user info. For example, UsersService might contain a method findById(userId) or checkUser(userId) that returns a dummy user object or Boolean. You can hard-code a small set of user IDs or assume any non-null ID is valid. This demonstrates how the Task service would interact with a user service in a real microservice environment.
* (Optional bonus) If time permits, demonstrate an **event or message** that might be sent to another microservice. For instance, log a message or simulate publishing an event when a task is assigned or completed (e.g., to notify a NotificationService). This is not required but shows awareness of eventual consistency in microservices.

## Authentication & Authorization

|  |  |
| --- | --- |
|  | Assume the Task service is behind an API gateway or uses JWT for auth in production. For this test, **simulate an authenticated context**: |

* Implement a simple **role-based guard** to protect the delete operation. Create a custom decorator (e.g., @Roles('admin')) to mark admin-only routes​, and a corresponding **RolesGuard** that checks the current user's role before allowing access. You can simulate obtaining the current user/role via a request header or a hard-coded value. For example, require an X-User-Role header and have the guard allow the request only if the role is "admin" for routes decorated with @Roles('admin'). Non-admin attempts should result in 403 Forbidden.
* (If you prefer, you can also implement a simple **AuthGuard** or middleware that rejects requests without a dummy auth token, but the main focus is demonstrating the guard/decorator mechanism for authorization.)

## Middleware/Interceptor:

|  |  |
| --- | --- |
|  | Include at least one of: |

* A global or route-specific **logging middleware** or **logging interceptor** that logs each incoming request or measures execution time.
* Or an **interceptor** that transforms responses (e.g., wraps successful responses in a common format) or errors. For example, you could use an interceptor to automatically convert any thrown error into a consistent JSON structure, or simply log the error.  
  (Only one of middleware or interceptor is required, but using both is fine. Choose what best showcases your understanding.)

## Exception Handling

|  |  |
| --- | --- |
|  | Use NestJS’s exception handling to produce proper HTTP responses: |

* Throw appropriate Nest HTTP exceptions (NotFoundException, BadRequestException, etc.) for error cases like "task not found", "user not found for assignment", or "invalid input". Nest will handle transforming these into HTTP error responses.
* Implement a **global exception filter** only if you need custom behavior; otherwise relying on Nest’s default exception handling is acceptable. If you do implement one (optional), for example, it could catch unhandled exceptions and log them or map them to a generic error response.

## NestJS Structure & SOLID

|  |  |
| --- | --- |
|  | Organize the code using NestJS’s module system and adhere to clean code principles: |

* Create separate **modules** for distinct features (e.g., TasksModule and UsersModule). Each module should declare its own controllers and providers.
* Use **controllers** for request handling and **providers (services)** for business logic. Ensure that controllers delegate to services, and services handle the core logic (e.g., interacting with the database).
* Leverage **dependency injection** to manage interactions: e.g., inject the TasksService into TasksController, inject UsersService into TasksService (to validate user IDs), and inject database access (Drizzle) into your services.
* Follow SOLID design principles (Single Responsibility, Dependency Inversion, etc.) in your solution – for example, use interfaces or abstraction for data access if appropriate, keep business logic separate from controllers, and make components testable.

## Database Layer (Drizzle + PostgreSQL)

|  |  |
| --- | --- |
|  | Use **Drizzle ORM** for all database interactions: |

* Define your database tables using Drizzle’s TypeScript schema definitions (e.g., define a tasks schema with columns). Drizzle is a type-safe, SQL-oriented ORM that uses schema objects and fluent queries.
* Configure a connection to a PostgreSQL database. You can use a connection URL (provided via an environment variable) or local config in the code. It’s fine to use a local Postgres instance or even an in-memory SQLite for this test (Drizzle supports SQLite too) if that’s easier, as long as your code clearly demonstrates how you would set up Drizzle for Postgres. For example, you might create a DatabaseModule or use AppModule to establish the DB connection and provide the Drizzle client via dependency injection (similar to how one would with TypeORM or Prisma).
* Use Drizzle in the TasksService (or in a dedicated repository provider) to perform CRUD operations: e.g., insert a new task, select tasks with optional filtering, update a task’s fields, and delete by ID. Make sure these calls are properly awaited and handled.
* **Mock data:** If needed, you can pre-populate the database with a couple of sample users/tasks or insert on the fly for testing. (For example, you might start with an empty tasks table. The first calls to create will populate it. For users, since user service is simulated, just ensure the UsersService knows about some valid user IDs.)

## Input / Output Expectations

|  |  |
| --- | --- |
|  | * Creating a task (POST /tasks) with valid data returns a **201 Created** response containing the new task (with its id and other fields). If the assignedUserId does not exist (and you choose to validate it), return a 400/404 error. * Getting tasks (GET /tasks) without filters returns a list of all tasks. Filtering by assignedUserId (GET /tasks?assignedUserId=XYZ) returns tasks that have that assignedUserId. If no tasks or no such user (depending on your approach), it can return an empty list (200 OK). * Getting a task by ID (GET /tasks/:id) returns the task data if found, or a **404** if not found. * Updating a task (PATCH /tasks/:id) returns the modified task. If the task doesn’t exist, respond with 404. If an invalid update is attempted (e.g., assign to non-existent user and you validate that), respond with 400 or 404 accordingly. * Deleting a task (DELETE /tasks/:id) with an admin role returns a 200/204 on success (no content needed or a confirmation message). If a non-admin attempts it, respond with **403 Forbidden** (the RolesGuard should handle this). If the task doesn’t exist, 404 Not Found (guard would check authorization first, then you handle existence).   Keep the JSON structure clean and use proper HTTP status codes for all responses. |