ChatGPT

Script Assist NestJS Technical Challenge

This challenge asks you to **build a backend API using NestJS** following the instructions in the provided repository. In simple terms, you will clone the GitLab repo (`scriptassist-nestjs-exercise`) and implement the required features by the deadline (Tuesday 7 pm UK time). The goal is to create a well-structured NestJS application with multiple modules (controllers, services, etc.) and demonstrate best practices. NestJS is "a progressive Node.js framework for building efficient, reliable and scalable server-side applications" [1], so expect to use its modular architecture, dependency injection, and TypeScript support.

Important points to cover include:

- **Main Functionality (Task Module)**: Implement a RESTful resource (often called "tasks" or similar) with standard CRUD operations. Use NestJS controllers and services to handle create/read/update/delete. Define DTO (Data Transfer Object) classes for input data and use Nest's `ValidationPipe` with [class-validator] to enforce rules (for example, `@IsString()`, `@IsNotEmpty()`, etc.) [2]. The API should allow filtering and pagination (e.g. via query parameters like `?page=1&limit=20`) so clients can fetch subsets of data. The example feedback suggests supporting filters and pagination with correct metadata (total count, pages, etc.) as a best practice.

- **Authentication (JWT with Refresh Tokens)**: Add endpoints for user signup and login that issue JWT access tokens (and refresh tokens for session renewal). Upon successful login, return a short-lived access token (e.g. expires in minutes) and a longer-lived refresh token. You should **securely store passwords** using a hashing library like `bcrypt` or `argon2` [3] and store refresh tokens (often hashed) on the server or database. Use NestJS's passport and JWT packages (`@nestjs/passport`, `@nestjs/jwt`) to implement the token strategy. A good guide notes that "Token-based authentication" involves users receiving a secure JWT containing their identity after login [4]. Make sure to handle token rotation/refresh properly (for example, an endpoint to exchange a valid refresh token for a new access token).

- **Authorization (Role-Based Access Control)**: Protect routes based on user roles (e.g. admin vs regular user). Use NestJS **guards** and custom decorators to enforce roles. For example, create a `@Roles([...])` decorator and a `RolesGuard` that checks the current user's role against the metadata on the handler [5]. In code you would annotate a route like `@Roles('admin')` to allow only admin users. Nest's official docs show using a `RolesGuard` that reads these roles with the `Reflector`, allowing only matching requests [5].

- **Background Processing (Queues)**: Some operations (like sending emails, processing overdue tasks, generating reports, etc.) should run **asynchronously**. Use a queue library like **Bull** or **BullMQ** with Redis. NestJS provides the `@nestjs/bull` / `@nestjs/bullmq` modules to easily integrate queues [6]. For example, set up a Bull queue in `AppModule` and create producers (to enqueue jobs) and consumers (processors). Queues help "smooth out processing peaks" and offload heavy work from request handlers [7].

- **Caching & Rate Limiting**: Improve performance with caching and protect endpoints with rate limiting. Use Nest's built-in **cache module** (backed by Redis or in-memory) to cache frequent queries [8] – this "allows quicker access to frequently used data" by storing it temporarily [8]. Also implement rate limiting (for instance using the Nest Throttler or middleware) to limit requests per IP or user. This prevents abuse (e.g. brute-force) and is part of a secure design.

- **Database & Data Layer**: Use a database (PostgreSQL, MySQL, Mongo, etc.) with an ORM/ODM (TypeORM, Sequelize, or Mongoose). Define entity models for your data (e.g. a `Task` entity) and use a repository pattern. Ensure queries are efficient (e.g. indexed, use pagination queries) and handle transactions if doing bulk ops. Keep database credentials/config in environment variables. The example feedback suggests using a repository/service pattern and even mentions efficient SQL for stats, so consider writing raw queries or query builders for any analytics endpoints.

- **API Documentation (Swagger/OpenAPI)**: Document your endpoints. NestJS supports Swagger via the `@nestjs/swagger` package. Install it and initialize `SwaggerModule` in `main.ts`, so that visiting `/api` or similar shows the interactive API docs [9]. Use decorators like `@ApiTags`, `@ApiResponse`, etc. on your controllers and DTOs to generate a clear OpenAPI spec. The docs note that Nest "provides a dedicated module which allows generating [OpenAPI] specification by leveraging decorators" [9]. Good documentation (including example requests/responses) will help evaluators understand your API.

- **Testing (Unit and End-to-End)**: Write automated tests. At minimum, create unit tests (with Jest) for your services and controllers to verify logic. NestJS integrates with Jest and **Supertest** out-of-the-box for HTTP testing [10]. For example, write e2e tests that spin up the app and hit endpoints (login, create task, etc.) to ensure they behave correctly. The framework's docs emphasize that Nest "provides integration with Jest and Supertest out-of-the-box" and scaffolds default tests [10]. Aim for decent coverage, especially around core features.

- **Security Best Practices**: Apply common security measures. Use validation pipes on all inputs (to avoid injection or bad data) [2]. Hash sensitive values (`bcrypt` for passwords) [3]. Use HTTPS in production and set HTTP headers (e.g. with Helmet) if possible. Enable CORS if needed. In summary, follow security guides: validate every request, handle errors gracefully (don't leak stack traces), and never log sensitive info.

- **Code Quality & Architecture**: Organize code into clear modules (e.g. `auth`, `tasks`, `users`, `common`, etc.). Keep controllers thin (delegate to services), and services should contain business logic. Use TypeScript's strong typing (enums for roles/statuses, interfaces, generics where helpful). Write clean, readable code with proper naming. The example feedback highlights modular architecture and separation of concerns; emulate that structure. Include any build/test scripts in `package.json`, and ensure the app can be started (e.g. `npm run start:dev`) after setting up the database.

- **Submission Details**: Finally, **submit your work** by pushing to a GitHub repository (even though the exercise repo is on GitLab, they ask for your GitHub link). Make sure to commit all code, update the README if needed with setup instructions, and provide any necessary database migration or seeding scripts. Remember the **deadline** (Tuesday 7 pm UK time) and send them the GitHub link.

Overall, think of this exercise as building a small production-ready NestJS backend. Key features include a well-designed CRUD module with validation [2], secure JWT auth with refresh tokens [4] [3], role guards [5], background job queues [6], caching [8], API docs [9], and tests [10]. The example feedback in the instructions highlights exactly these areas, so covering them thoroughly is important. Code readability, best practices, and completeness (including error handling and logging) are also crucial.

---

[1] NestJS - A progressive Node.js framework
https://nestjs.com/

[2] Validation | NestJS - A progressive Node.js framework
https://docs.nestjs.com/techniques/validation

[3] Encryption and Hashing | NestJS - A progressive Node.js framework
https://docs.nestjs.com/security/encryption-and-hashing

[4] NestJS JWT Authentication with Refresh Tokens Complete Guide - Elvis Duru
https://www.elvisduru.com/blog/nestjs-jwt-authentication-refresh-token

[5] Guards | NestJS - A progressive Node.js framework
https://docs.nestjs.com/guards

[6] [7] Queues | NestJS - A progressive Node.js framework
https://docs.nestjs.com/techniques/queues

[8] Caching | NestJS - A progressive Node.js framework
https://docs.nestjs.com/techniques/caching

[9] OpenAPI (Swagger) | NestJS - A progressive Node.js framework
https://docs.nestjs.com/recipes/swagger

[10] Testing | NestJS - A progressive Node.js framework
https://docs.nestjs.com/fundamentals/testing