Backend API Design & Implementation Challenge

<u>**Backend Challenge**</u>

Language & Stack Requirement: Node.js with TypeScript

Adopt a NoSQL-style schema (e.g., document-oriented or in-memory model); no actual database setup is required.

Code Quality: Must meet industry standards (readability, modularity, type safety, linting, consistent style, maintainability, meaningful commit hygiene).

Expected effort & timeline: Please submit your completed solution within 5 **calendar days** of receiving the prompt. If you need a faster turnaround (e.g., for senior/expedited screening), an **accelerated 72-hour** option is acceptable.

Overview

You are to design and implement a small backend service exposing four core endpoints using **Node.js with TypeScript**. The purpose is to evaluate your ability to build production-quality APIs: designing data models, enforcing invariants, handling concurrency, securing access, managing file uploads, and explaining your decisions.

Deliverables (to be submitted as a repository or ZIP):

- Implementing the four endpoints.
- Data model/schema and any persistence layer.
- Authentication and authorization.
- Input validation, error handling, and logging.
- API documentation (README + usage examples).
- Short design write-up explaining design decisions, tradeoffs, and scalability.
- At least one custom business rule you invented, documented, and justified.

*You may use any libraries/frameworks/storage in the Node.js/TypeScript ecosystem; you must **justify your technology choices**.*

Core Functional Requirements

1. GET endpoint with complex formatting

- Resource: e.g., a **Report** containing nested child collections (entries, comments, metadata, we need at least 10 fields).
- Include computed/aggregated fields (e.g., totals, derived status, trend indicators).
- Support multiple output shapes:
  - Default: rich hierarchical JSON with nested arrays.
  - Alternate view: a "compact summary" or flattened human-readable variant triggered by a query parameter.
- Selective expansion/inclusion of subfields via query parameters (e.g.,? include=entries,metrics).
- Pagination for large nested lists (e.g., entries with page & size).
- Filtering/sorting inside nested collections (e.g., recent or high-priority entries first).

2. PUT endpoint for editing

- Update the same resource with both full and partial semantics.
- Guarantee **idempotency** (e.g., idempotency keys or safe PUT semantics).
- Validate incoming payloads; return structured error responses with codes, messages, and field-level details.
- Audit/log changes meaningfully: who changed what and when (store before/after or equivalent metadata).

3. POST endpoint

- Create a new resource (e.g., a Report).
- Server-side generated a unique identifier.
- Enforce business invariants (e.g., no duplicate business key).
- Sanitize and validate input.
- Return proper HTTP semantics (201 Created, Location header, representation).
- Trigger an **asynchronous side effect** (e.g., enqueue a background job, send notification, invalidate cache) with clear failure handling (retry/backoff, dead-letter, compensating marker).

4. File upload endpoint

- Accept a file upload tied to an existing resource (e.g., POST /reports/{id}/attachment).
- Support multipart or signed upload flow.
- Enforce file type and size restrictions.
- Store via an abstracted file storage layer (local disk, cloud object store, etc.) and maintain a secure reference.
- Provide safe download access (e.g., signed URLs, expiring tokens).
- *(Optional in implementation)* Describe how you would integrate malware/virus scanning in production.

Non-functional Requirements

Security

- Authentication (e.g., JWT, API key) with at least two roles (e.g., reader vs editor) and enforced authorization logic.
- Input validation to prevent injection and malformed data.
- Explicit assumptions about transport security (HTTPS) even if simulated.

Observability

- Structured request-aware logging (including trace/request IDs).
- Consistent error response schema.

Scalability

- Support horizontal scaling (stateless vs stateful separation).
- Thoughtful data access (indexes, pagination).

Code Quality

Candidates must **demonstrate** industry-standard maintainability:

- Modular separation of concerns.

- Meaningful naming; no dead/commented-out cruft.
- Use of TypeScript type annotations and validation schemas.
- Clean commit history showing logical steps. We will validate your commit history.
- A section in the design write-up titled **“Code Quality Practices”** covering linters, static analysis, testing philosophy, type safety, and other quality mechanisms.

Deep-Probing Design

1. Ambiguity That Must Be Resolved

Core domain details are intentionally underspecified. Candidates must *explicitly state and justify assumptions*. Superficial treatment of unspecified parts will reduce score.

2. Design Justification Required

Every major decision (frameworks/libraries, schema, concurrency control approach, auth model, file storage mechanism, asynchronous side effect handling) must be justified in context. Generic boilerplate rationale is insufficient.

3. Custom Business Rule

Introduce at least one non-trivial business rule of your own design (e.g., “Reports in status X can only be edited by role Y if condition Z is met”). Document it and explain its impact on validation, API behavior, and data modeling.

4. Evolving Spec Mentality

Describe how the design can absorb changes (e.g., new computed metrics, changed expansion semantics, additional views) with minimal rework.

Candidate Prompt

Following can be an example (Just consider following as an example, you are not required to follow the report logic):

Implement the following:

1. **GET** /reports/{id}: Return the report including nested entries, computed metrics, and allow an alternate summary view.
2. **PUT** /reports/{id}: Edit the report, supporting full/partial updates, idempotency, and optimistic concurrency control.
3. **POST** /reports: Create a new report, enforcing uniqueness, sanitizing input, and triggering an asynchronous side effect with failure handling.
4. **POST** /reports/{id}/attachment: Upload a file tied to a report, validating content, storing securely, and exposing safe access.

You must authenticate requests, enforce role-based authorization, log changes, and handle errors consistently. The domain is intentionally underspecified; clearly state and justify your assumptions. Include at least one custom business rule you invented.

Important: This is not a copy/paste exercise. Show your thinking: explain tradeoffs, document your code quality practices, and justify every significant design decision. Use of AI tools is allowed, but raw copy/paste won't be enough; demonstrable understanding, adaptation to ambiguity, and personalization are required to score highly.

Deliverables

- Source code repository or ZIP with history (meaningful commits).
- **README** including:
  - Setup and run instructions.
  - Authentication usage (test tokens or how to obtain credentials).
  - Examples (curl or code) for each endpoint.
  - Description of the custom business rule added.
- **design.md** (or equivalent) covering:
  - Schema and data model.
  - Authentication/authorization model.
  - Concurrency control approach.
  - File storage/access security.

    - Asynchronous side effect strategy and failure handling.
    - Code quality practices.
    - Scaling and observability considerations.
- *(Optional)* Extensions and next steps for production-grade evolution.
- What are the required and optional fields of a Report and its nested entries?
- What roles exist and what permissions do they have over reading/updating reports and attachments?
- How long should download links be valid and who may use them?
- Provide the payloads as required for your business logic