Mid-Level Node.js Developer Technical Test: Full Stack Collaboration Platform (Backend Focus)

Overview

You are required to build a back-end API for a collaborative project management platform, simulating a Trello-like task tracker, secured by a complete JWT authentication and authorization system. This test is designed to evaluate your skills in secure authentication, complex authorization models, **PostgreSQL data modeling**, mixed API architecture (REST/GraphQL), real-time updates (Subscriptions), and robust logging.

Architecture Mandates

- **Runtime: Bun** (for both runtime execution and package management).
- **Framework:** Express.js (with Bun).
- **Database: PostgreSQL** (raw sql). Candidates should specify their choice and set up the environment (e.g., Docker Compose or Neon DB).
- **API:** Primarily GraphQL (Apollo Server recommended). Must use **REST** for specific token-related endpoints.
- **Authentication:** JSON Web Tokens (JWT) with separate Access and Refresh Tokens.
- **Logging:** Must implement secure dual-logging (File + Database) across multiple required categories.

Part 1: Authentication & User Management (Mixed Endpoints)

Implement the following endpoints using the specified API types.

Feature	API Type	Notes
Register	GraphQL Mutation	Standard user creation.

Login	**REST (POST)**	Returns `accessToken` (short-lived) and `refreshToken` (long-lived, HTTP-only cookie recommended).
Logout	**REST (POST)**	Invalidates the `refreshToken` server-side.
Refresh Token	**REST (POST)**	Accepts `refreshToken`, issues a new `accessToken`.
Forgot Password	GraphQL Mutation	Requires email to send a unique reset link/code (mock the email sending).
Update Password	GraphQL Mutation	Allows authenticated users to change their own password.

Admin-Specific Functionality

The application must support the 'ADMIN' status.

Feature (Admin Only)	GraphQL Mutation	Notes
User Ban/Unban	Mutation	Prevents/allows user login by setting their Global Status to `BANNED` or `ACTIVE`. Must trigger a Security Log entry.
Admin Reset Password	Mutation	Force-resets a user's password (e.g., if they are locked out). Requires admin credentials.

Device and Session Management

- **Device Tracking:** On successful login, the system must save device metadata associated with the `refreshToken` to a dedicated table/model (`UserDevices`).
- **Metadata Fields:** Must include `IPAddress`, `userAgent`, `loginTime`, and an `isRevoked` flag for session termination.

Part 2: Complex Authorization & Workspaces

The system must support an authorization hierarchy based on **Workspaces** and **Projects**.

Data Model Requirements (PostgreSQL)

1. **User:** Basic auth fields, **Global Status (Enum: `ACTIVE`, `BANNED`, `ADMIN`).**
2. **Workspace:** Name, list of member references, and their associated role.
3. **Workspace Role:** Must be implemented as an Enum or fixed set:
    - **Owner:** Full CRUD rights on the Workspace, can add/remove members, and assign roles.
    - **Member:** Can create/edit/delete Projects and Tasks within the Workspace.
    - **Viewer:** Can only read/view Projects and Tasks.
4. **Project Role (NEW):** Must be implemented as an Enum or fixed set, granting granular permissions within a specific Project:
    - **Project Lead:** Can manage project membership/roles, edit all tasks, and delete the project.
    - **Contributor:** Can create, edit, and update the status of tasks assigned to them or unassigned tasks.
    - **Project Viewer:** Read-only access to all project tasks and details.

Candidate Flexibility: *You have the freedom to adjust, expand, or refactor these data models (tables, columns, foreign keys, etc.) as needed, provided that the final functional requirements for authorization and data integrity are fully met.*

Workspace GraphQL Endpoints

- **`createWorkspace`:** User who creates it is automatically assigned the `Owner` role.
- **`addWorkspaceMember`:** Requires `Owner` role. Assigns a default `Member` role.
- **`removeWorkspaceMember`:** Requires `Owner` role.
- **`updateWorkspaceMemberRole`:** Requires `Owner` role. Cannot change the role of the Owner.
- **`getWorkspace`:** Requires at least `Viewer` role membership in that Workspace.
- **Admin View:** Implement a query (e.g., `getAllWorkspaces`) that is **only accessible by ADMIN users**, allowing them to view **all workspaces and their full member lists** across the entire application.

Evaluation Note: *The quality and precision of the* ***granular permission checks*** *(Owner > Lead > Contributor > Viewer) will be the most heavily weighted factor in the review.*

Part 3: Task Tracker, Notifications & Real-Time

Within a Workspace, users can create Projects, and Projects contain Tasks.

Data Model Requirements (PostgreSQL)

1. **Project:** Name, associated `WorkspaceId`. Project membership must be tracked, and each member must have a **Project Role** (see Part 2). Project members must already be members of the parent Workspace. A dedicated **ProjectMembership** model/table is highly recommended to manage the many-to-many relationship and store the role.
2. **Task:** Title, Description, Status (e.g., `TODO`, `IN_PROGRESS`, `DONE`), **`assignedToIds` (List of User IDs)**, `ProjectId`.
3. **Notification (NEW):** Implement a model to track user notifications.
    - Fields: `title`, `body`, `recipientId` (User ID), `status` (Enum: `DELIVERED`, `SEEN`), `relatedEntityId` (e.g., Task ID).

Task Tracker & Notifications GraphQL Endpoints

Feature	GraphQL Mutation/Query	Notes
Project CRUD	Mutations	Standard mutations for creating, updating, and deleting projects. (Requires Workspace `Member` or `Owner` role).
Update Project Member Role (NEW)	Mutation	Allows Project Leads and Workspace Owners to change a user's role within a specific project.
Task CRUD	Mutations	Standard mutations for creating, updating, and deleting tasks. (Requires Project `Contributor` or `Project Lead` role).

Task Assignment & Notification Trigger	Mutation	When a Task is assigned or re-assigned (via Task update/creation), a `DELIVERED` **Notification entry must be created** for all newly assigned users.
Notification Management	Mutations	Mutations to mark a notification as `SEEN`.

Real-Time Requirement: GraphQL Subscriptions (CRITICAL)

When the **status of any Task is changed** within a Workspace, all currently connected users who are **members of that Workspace** must receive a real-time update via a GraphQL Subscription.

- **Subscription:** `taskStatusUpdated(workspaceId: ID!): Task`
- **The subscription payload should only contain necessary task data and must be filtered by `workspaceId`.**

Part 4: Secure Dual Logging (Security Requirement)

Implement a secure logging system that writes logs to **two separate destinations** (File and Database). This system must categorize and handle **User Logs, System Logs, and Activity Tracker** events. This should be handled by a dedicated service/utility (e.g., using Winston).

Destination 1: Secure Log File (Production Monitoring)

- Log all activity to a local file (`audit.log`).
- Include timestamp, severity level, message, and the responsible **User ID**.

Destination 2: Database Audit Collection (Compliance)

Create a dedicated database table (`auditLogs`) for security-sensitive events and key activity.

Events to Log in the DB (Expanded Scope):

1. **Authentication Failures:** Failed login attempts (include IP address if possible). (System/Security Log)

2. **Admin Actions:** Every time an Admin uses `userBan/Unban` or `adminResetPassword`. The log must include the target user ID and the admin user ID. (Security Log)
3. **Critical Errors:** All exceptions and unhandled errors. (System Log)
4. **Task Lifecycle:** Task status changes (`TODO` -> `IN_PROGRESS` -> `DONE`). (Activity Tracker)
5. **Project/Workspace Management:** Creation, deletion, and membership changes for Workspaces and Projects. (Activity Tracker)
6. **Session Management:** Successful login/logout and device revocation. (User Log)

Log Structure (Both File and DB)

Field	Description
`timestamp`	UTC Date/Time of event.
`level`	`info`, `warn`, `error`, `security`.
`userId`	ID of the authenticated user performing the action (if applicable).
`ipAddress`	User's IP address (for auth attempts/session logs).
`action`	A concise string describing the operation (e.g., `LOGIN_FAILURE`, `USER_BANNED`, `TASK_STATUS_UPDATE`).
`details`	A JSON object containing specific context (e.g., `{ targetUserId: '...' }`, `{ newStatus: 'DONE' }`).

Part 5: Testing and Deployment (New Mandatory Requirements)

Mandatory Testing (Unit & E2E)

The submission must include a comprehensive test suite to ensure code correctness and prevent regressions.

1. **Unit Tests:** Implement unit tests for all critical functions, focusing on **utility services** (e.g., token generation, hashing) and **core business logic** (e.g., notification creation, logging service).
2. **End-to-End (E2E) Tests:** Implement E2E tests for the most security-sensitive and complex workflows:
    - **Authentication Flow:** Successful registration, login, token refresh, and logout.
    - **Authorization Check:** A test that verifies a non-member user **cannot** view a restricted workspace.

Continuous Integration / Continuous Deployment (CI/CD)

Demonstrate CI/CD readiness by including a configuration file for a common CI tool (e.g., **GitHub Actions, GitLab CI, or Jenkinsfile**). This pipeline must define the steps necessary to:

1. **Install** dependencies (`bun install`).
2. **Run** the full test suite (Unit and E2E).
3. **Build** the application (if applicable).
4. **Lighthouse/Lint** the code (highly recommended).

Bonus Challenge: AI Integration (Highly Valued)

Demonstrate capability to integrate and manage external AI services using the **Gemini API** within your GraphQL resolvers.

Feature	GraphQL Mutation/Query	Notes
Summarize Task	Query	Accepts a long task description string and uses the AI to return a concise, 1-2 sentence summary.

Generate Tasks from Prompt	Mutation	Accepts a high-level prompt (e.g., "Create a plan for launching a website") and uses the AI to return a structured list of potential tasks. The resolver should then **persist** these generated tasks into the specified Project in the PostgreSQL database.

Extra Credit & Differentiators (Go Overboard!)

Due to the high volume of applications, demonstrating initiative and advanced architectural planning beyond the core requirements is highly encouraged and will significantly increase your chance of moving to the next phase.

Suggestions for Differentiating Features:

- **VAPID(firebase) Web Push Notifications:** Implement real-time user notification delivery outside the application (e.g., when a user is assigned a task).
- **Rate Limiting & Throttling:** Implement robust rate limiting on public endpoints (e.g., login, registration) to prevent brute-force attacks.
- **Graceful Shutdown:** Implement a process for the application to handle signals (like `SIGTERM`) to shut down gracefully, closing database connections and logging the shutdown.
- **Dockerized Deployment:** Provide a complete `Dockerfile` and `docker-compose.yml` that bundles the entire application (Node.js/Bun) and the PostgreSQL database, making setup trivial.

Submission, Documentation & Next Steps

Timeline and Submission Requirements (CRITICAL)

- **Deadline: October 27, 4:00 AM local time.** This provides a 3-day window for focused work.
- **Completion Expectation:** Due to the comprehensive nature of the task, **full completion is not expected, however, demonstrating ability to go above and beyond the required features is essential for progression.** Focus on delivering a solid, working core that demonstrates robust architecture, security best practices (JWT, Dual Logging), **CI/CD readiness**, and complex authorization logic. **The evaluation will heavily prioritize the quality and correctness of the granular permission checks and the test coverage.**
- **Scope:** The primary focus must be on the **backend API**.

    - **Backend (MANDATORY):** Must be fully functional (Bun, Postgres, Auth, GQL logic).
    - **Frontend Demonstration (MINIMAL):** The system must be demonstrated. Include a **minimal frontend client** (e.g., a simple React/Vue app, or detailed cURL/Postman/Insomnia collection files) that can demonstrate **Login/Logout**, **Task Creation**, and the **GraphQL Subscription** functionality.
- **Version Control:** We expect a **clear, detailed commit history (aim for 20+ meaningful, small commits)** to demonstrate an iterative and organized development process.
- **Demonstration Video (MANDATORY):** The candidate should submit the complete source code, including instructions on how to set up and run the application (e.g., a README.md and sample queries/mutations), and a **brief (2-5 minute) video walkthrough** demonstrating the setup and key functionalities achieved (e.g., Admin banning a user, a task status update triggering the GQL Subscription).

Documentation Requirement (Mandatory)

A comprehensive `README.md` is required. It should include:

1. Clear setup and run instructions (for both the Bun app and PostgreSQL).
2. A brief explanation of your overall architecture and why you chose your data models.
3. A summary of all implemented features and any limitations.

AI Usage Policy (Clarification)

- **Allowed:** AI tools (like Gemini/ChatGPT) are permitted for generating boilerplate code, setup scripts (e.g., Docker files), and the minimal frontend client.
- **Minimized/Understood:** Candidates must **minimize AI usage** on the core backend logic, especially authorization checks, database queries, and logging utilities. **If asked in the final presentation, candidates must be able to fully explain every line of their submitted backend code without hesitation.**

Next Phase: Code Presentation

If you are selected to move forward, the next stage will be a **live, virtual code presentation (Zoom)** with our technical team. You will be expected to:

- Walk through your solution's structure and data model.
- Explain the architecture and logic behind your **granular permission checks**.
- Justify your design choices and demonstrate a deep understanding of the entire codebase.