

Transcript B

Test Environment

Date: March 2026

Model: Claude

Interface: Claude API

Continuity Layer: Calmkeep (visit @ Calmkeep.ai)

This transcript was generated using Claude through the Claude API with the Calmkeep continuity layer enabled. The prompt sequence was identical to the prompts used for Transcript A in the Claude App.

USER: I want to build a multi-tenant SaaS task management API. Tech stack should be:

- Node.js with TypeScript
- PostgreSQL database
- Express for the API
- JWT for authentication

Set up the initial project structure and explain the folder organization you're recommending.

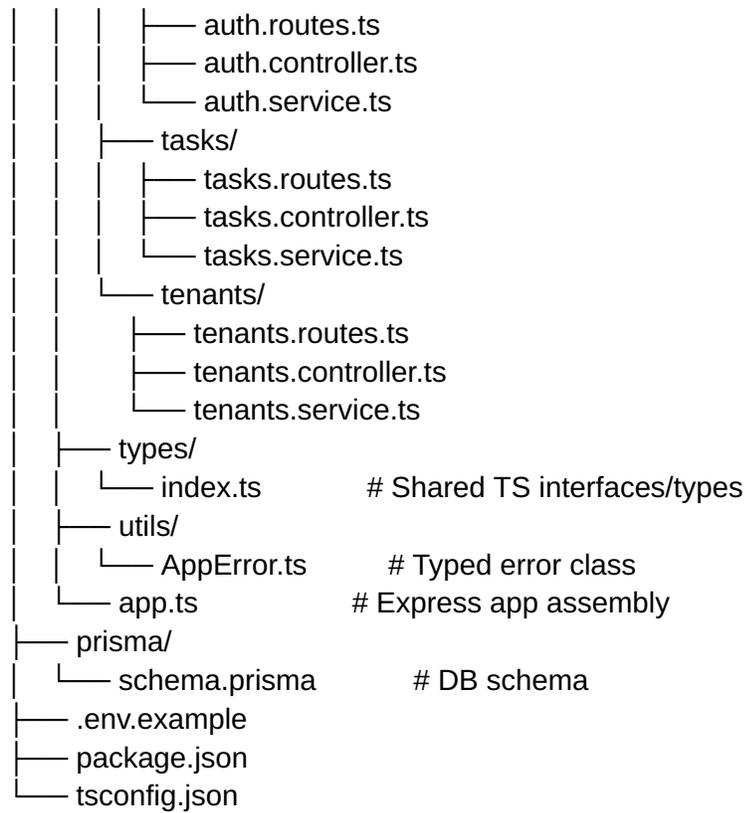
Multi-Tenant SaaS Task Management API – Initial Setup

Folder Structure

...

task-management-api/

```
|— src/
|   |— config/
|   |   |— database.ts      # DB connection/pool config
|   |   |— env.ts          # Validated env vars (fail-fast)
|   |— middleware/
|   |   |— authenticate.ts  # JWT verification
|   |   |— tenantContext.ts # Extracts + validates tenant
|   |   |— errorHandler.ts # Central error handler
|   |— modules/
|   |   |— auth/
```



Why This Structure

| Decision | Rationale |

|---|---|

| **Modules over layers** | Auth, tasks, tenants are cohesive vertical slices. Adding a new feature means adding one folder, not touching 4 separate layer folders. |

| **Middleware is flat** | Middleware is cross-cutting, not feature-specific. Keeping it separate avoids circular imports. |

| **Config is isolated** | Env vars are validated once at startup. Everything else imports from `config/`, never from `process.env` directly. |

| **`types/` is shared** | Interfaces used across modules (e.g., `AuthenticatedRequest`) live here to avoid circular dependencies. |

Implementation

`package.json`

```
```json
{
 "name": "task-management-api",
 "version": "1.0.0",
 "scripts": {
 "dev": "ts-node-dev --respawn --transpile-only src/app.ts",
 "build": "tsc",
 "start": "node dist/app.js",
 "prisma:generate": "prisma generate",
 "prisma:migrate": "prisma migrate dev"
 },
 "dependencies": {
 "@prisma/client": "^5.10.0",
 "bcryptjs": "^2.4.3",
 "express": "^4.18.3",
 "jsonwebtoken": "^9.0.2",
 "zod": "^3.22.4"
 },
 "devDependencies": {
 "@types/bcryptjs": "^2.4.6",
 "@types/express": "^4.17.21",
 "@types/jsonwebtoken": "^9.0.5",
 "@types/node": "^20.11.0",
 "prisma": "^5.10.0",
 "ts-node-dev": "^2.0.0",
 "typescript": "^5.3.3"
 }
}
```
```

`tsconfig.json`

```
```json
{
 "compilerOptions": {
 "target": "ES2020",
 "module": "commonjs",
 "lib": ["ES2020"],
 "outDir": "./dist",
 }
}
```

```
"rootDir": "./src",
"strict": true,
"esModuleInterop": true,
"skipLibCheck": true,
"forceConsistentCasingInFileNames": true,
"resolveJsonModule": true
},
"include": ["src/**/*"],
"exclude": ["node_modules", "dist"]
}
...

```

---

```
`src/config/env.ts`
```

```
``typescript
```

```
import { z } from "zod";
```

```
const envSchema = z.object({
 NODE_ENV: z.enum(["development", "production", "test"]).default("development"),
 PORT: z.string().default("3000"),
 DATABASE_URL: z.string().url(),
 JWT_SECRET: z.string().min(32, "JWT_SECRET must be at least 32 characters"),
 JWT_EXPIRES_IN: z.string().default("7d"),
});
```

```
const parsed = envSchema.safeParse(process.env);
```

```
if (!parsed.success) {
 console.error("❌ Invalid environment variables:");
 console.error(parsed.error.flatten().fieldErrors);
 process.exit(1); // Fail fast — never start with bad config
}
```

```
export const env = parsed.data;
```

```
...
```

---

```
`src/config/database.ts`
```

```
``typescript
import { PrismaClient } from "@prisma/client";

// Single shared instance – prevents connection pool exhaustion
const prisma = new PrismaClient({
 log: process.env.NODE_ENV === "development" ? ["query", "error"] : ["error"],
});

export default prisma;
``
```

---

```
`src/types/index.ts`
```

```
``typescript
import { Request } from "express";

export interface TenantContext {
 tenantId: string;
}

export interface JwtPayload {
 userId: string;
 tenantId: string;
 email: string;
}

// Extends Express Request so middleware can attach typed data
export interface AuthenticatedRequest extends Request {
 user: JwtPayload;
 tenant: TenantContext;
}
``
```

---

```
`src/utils/AppError.ts`
```

```
``typescript
export class AppError extends Error {
 constructor(
```

```
 public message: string,
 public statusCode: number,
 public isOperational = true // Distinguishes expected errors from bugs
) {
 super(message);
 Object.setPrototypeOf(this, AppError.prototype);
 }
}
...

```

---

```
`src/middleware/errorHandler.ts`
```

```
``typescript
```

```
import { Request, Response, NextFunction } from "express";
import { AppError } from "../utils/AppError";
```

```
export function errorHandler(
 err: Error,
 req: Request,
 res: Response,
 next: NextFunction // Required 4th param for Express to recognize as error handler
): void {
 if (err instanceof AppError) {
 res.status(err.statusCode).json({
 status: "error",
 message: err.message,
 });
 return;
 }
}
```

```
// Unhandled/unexpected errors – don't leak internals
console.error("Unexpected error:", err);
res.status(500).json({
 status: "error",
 message: "Internal server error",
});
}
...

```

---

```
`src/app.ts`
```

```
``typescript
```

```
import express from "express";
import { env } from "../config/env";
import { errorHandler } from "../middleware/errorHandler";
```

```
const app = express();
```

```
app.use(express.json());
```

```
// Health check – useful for load balancers and uptime monitors
```

```
app.get("/health", (req, res) = {
 res.json({ status: "ok", environment: env.NODE_ENV });
});
```

```
// Routes will be mounted here as modules are built
```

```
// app.use("/api/v1/auth", authRoutes);
```

```
// app.use("/api/v1/tasks", authMiddleware, tenantMiddleware, taskRoutes);
```

```
app.use(errorHandler);
```

```
app.listen(env.PORT, () = {
```

```
 console.log(`🚀 Server running on port ${env.PORT} [${env.NODE_ENV}]`);
});
```

```
export default app;
```

```
``
```

```

```

```
`.env.example`
```

```
``bash
```

```
NODE_ENV=development
```

```
PORT=3000
```

```
DATABASE_URL="postgresql://user:password@localhost:5432/taskdb"
```

```
JWT_SECRET=your-super-secret-key-minimum-32-characters-long
```

```
JWT_EXPIRES_IN=7d
```

```
``
```

--

```
`prisma/schema.prisma`
```

```
``prisma
```

```
generator client {
 provider = "prisma-client-js"
}
```

```
datasource db {
 provider = "postgresql"
 url = env("DATABASE_URL")
}
```

```
model Tenant {
 id String @id @default(cuid())
 name String
 slug String @unique
 createdAt DateTime @default(now())
 updatedAt DateTime @updatedAt
```

```
 users User[]
 tasks Task[]
}
```

```
model User {
 id String @id @default(cuid())
 email String
 passwordHash String
 createdAt DateTime @default(now())
 updatedAt DateTime @updatedAt
```

```
 tenantId String
 tenant Tenant @relation(fields: [tenantId], references: [id])
```

```
 tasks Task[]
```

```
 @@unique([email, tenantId]) // Same email can exist across different tenants
}
```

```
model Task {
 id String @id @default(cuid())
```

```
title String
description String?
status TaskStatus @default(TODO)
createdAt DateTime @default(now())
updatedAt DateTime @updatedAt

tenantId String
tenant Tenant @relation(fields: [tenantId], references: [id])

assigneeId String?
assignee User? @relation(fields: [assigneeId], references: [id])

@@index([tenantId]) // Critical – every query will filter by tenantId
}
```

```
enum TaskStatus {
 TODO
 IN_PROGRESS
 DONE
}
...

```

### ## Key Multi-Tenancy Design Decisions

```
...
Every DB model has tenantId
 |
 ▼
Middleware extracts tenant from JWT
 |
 ▼
Services always filter WHERE tenantId = req.tenant.tenantId
 |
 ▼
No query ever runs without a tenant scope
...

```

**\*\*The `@@unique([email, tenantId])` on User\*\*** means the same email address can register independently in different tenants – which is correct SaaS behavior.

**\*\*The `@@index([tenantId])` on Task\*\* is non-negotiable for performance. Every task query will filter by tenant, so PostgreSQL needs this index.**

---

## ## Getting Started

```
``bash
npm install
cp .env.example .env
Edit .env with your values

npx prisma migrate dev --name init
npx prisma generate
npm run dev
``
```

---

## ## What's Scoped for Next Steps

The natural build order from here:

1. **\*\*Auth module\*\*** – register, login, JWT issuance
2. **\*\*Tenant middleware\*\*** – extract and validate `tenantId` from the JWT on every protected route
3. **\*\*Tasks CRUD\*\*** – always scoped to `req.tenant.tenantId`

**USER: Design the database schema for this. We need:**

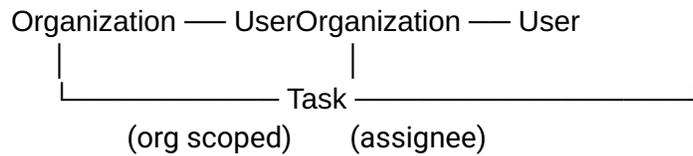
- **Users (with email, password hash, role)**
- **Organizations (multi-tenant structure)**
- **Tasks (belonging to organizations)**
- **User-Organization relationships (users can belong to multiple orgs)**

**Show me the SQL schema and explain the relationships.**

```
Database Schema Design
```

## ## Entity Relationship Overview

...



...

A User can belong to many Organizations. An Organization has many Users. Tasks belong to one Organization and optionally one User (assignee). The `UserOrganization` join table is the multi-tenancy membership record – it carries the role.

---

## ## Updated `prisma/schema.prisma`

This replaces the schema from the previous step entirely.

```
``prisma
generator client {
 provider = "prisma-client-js"
}

datasource db {
 provider = "postgresql"
 url = env("DATABASE_URL")
}

// _____
// ENUMS
// _____

enum OrgRole {
 OWNER // Full control – can delete org, manage billing
 ADMIN // Can manage members and tasks
 MEMBER // Can create and work on tasks
}

enum TaskStatus {
 TODO
 IN_PROGRESS
 DONE
```

```

}

enum TaskPriority {
 LOW
 MEDIUM
 HIGH
}

// _____
// MODELS
// _____

model Organization {
 id String @id @default(cuid())
 name String
 slug String @unique // URL-safe identifier e.g. "acme-corp"
 createdAt DateTime @default(now())
 updatedAt DateTime @updatedAt

 memberships UserOrganization[]
 tasks Task[]
}

model User {
 id String @id @default(cuid())
 email String @unique // Global uniqueness – one account, many orgs
 passwordHash String
 firstName String
 lastName String
 createdAt DateTime @default(now())
 updatedAt DateTime @updatedAt

 memberships UserOrganization[]
 assignedTasks Task[] @relation("TaskAssignee")
 createdTasks Task[] @relation("TaskCreator")
}

model UserOrganization {
 id String @id @default(cuid())
 role OrgRole @default(MEMBER)
 joinedAt DateTime @default(now())
}

```

```

userId String
user User @relation(fields: [userId], references: [id], onDelete: Cascade)

organizationId String
organization Organization @relation(fields: [organizationId], references: [id], onDelete:
Cascade)

@@unique([userId, organizationId]) // A user can only have one role per org
@@index([organizationId]) // Look up all members of an org efficiently
@@index([userId]) // Look up all orgs a user belongs to efficiently
}

model Task {
 id String @id @default(cuid())
 title String
 description String?
 status TaskStatus @default(TODO)
 priority TaskPriority @default(MEDIUM)
 dueDate DateTime?
 createdAt DateTime @default(now())
 updatedAt DateTime @updatedAt

 organizationId String
 organization Organization @relation(fields: [organizationId], references: [id], onDelete:
Cascade)

 assigneeId String?
 assignee User? @relation("TaskAssignee", fields: [assigneeId], references: [id], onDelete:
SetNull)

 creatorId String
 creator User @relation("TaskCreator", fields: [creatorId], references: [id], onDelete: Restrict)

 @@index([organizationId]) // Every task query filters by org – mandatory
 @@index([organizationId, status]) // Common filter: tasks in org by status
 @@index([assigneeId]) // Look up tasks assigned to a user
}
...

--

Equivalent Raw SQL

```

For reference and direct PostgreSQL use:

```
```sql
```

```
--  
-----  
-- ENUMS  
--  
-----
```

```
CREATE TYPE org_role AS ENUM ('OWNER', 'ADMIN', 'MEMBER');  
CREATE TYPE task_status AS ENUM ('TODO', 'IN_PROGRESS', 'DONE');  
CREATE TYPE task_priority AS ENUM ('LOW', 'MEDIUM', 'HIGH');
```

```
--  
-----  
-- ORGANIZATIONS  
--  
-----
```

```
CREATE TABLE organizations (  
  id    TEXT    PRIMARY KEY DEFAULT gen_random_uuid(),  
  name  TEXT    NOT NULL,  
  slug  TEXT    NOT NULL UNIQUE,  
  created_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),  
  updated_at TIMESTAMPTZ NOT NULL DEFAULT NOW()  
);
```

```
--  
-----  
-- USERS  
--  
-----
```

```
CREATE TABLE users (  
  id      TEXT    PRIMARY KEY DEFAULT gen_random_uuid(),  
  email   TEXT    NOT NULL UNIQUE,  
  password_hash TEXT NOT NULL,  
  first_name TEXT NOT NULL,  
  last_name TEXT NOT NULL,  
  created_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),  
  updated_at TIMESTAMPTZ NOT NULL DEFAULT NOW()  
);
```

```
--  
-----  
-- USER - ORGANIZATION MEMBERSHIP
```

```
CREATE TABLE user_organizations (  
  id      TEXT      PRIMARY KEY DEFAULT gen_random_uuid(),  
  user_id TEXT      NOT NULL REFERENCES users(id)      ON DELETE CASCADE,  
  organization_id TEXT NOT NULL REFERENCES organizations(id) ON DELETE CASCADE,  
  role    org_role  NOT NULL DEFAULT 'MEMBER',  
  joined_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
```

```
  CONSTRAINT uq_user_org UNIQUE (user_id, organization_id)  
);
```

```
CREATE INDEX idx_user_org_org_id ON user_organizations(organization_id);  
CREATE INDEX idx_user_org_user_id ON user_organizations(user_id);
```

```
-- TASKS
```

```
CREATE TABLE tasks (  
  id          TEXT      PRIMARY KEY DEFAULT gen_random_uuid(),  
  organization_id TEXT NOT NULL REFERENCES organizations(id) ON DELETE CASCADE,  
  creator_id  TEXT      NOT NULL REFERENCES users(id)      ON DELETE RESTRICT,  
  assignee_id TEXT      REFERENCES users(id)      ON DELETE SET NULL,  
  title       TEXT      NOT NULL,  
  description  TEXT,  
  status      task_status NOT NULL DEFAULT 'TODO',  
  priority    task_priority NOT NULL DEFAULT 'MEDIUM',  
  due_date    TIMESTAMPTZ,  
  created_at  TIMESTAMPTZ NOT NULL DEFAULT NOW(),  
  updated_at  TIMESTAMPTZ NOT NULL DEFAULT NOW()  
);
```

```
CREATE INDEX idx_tasks_org_id    ON tasks(organization_id);  
CREATE INDEX idx_tasks_org_status ON tasks(organization_id, status);  
CREATE INDEX idx_tasks_assignee_id ON tasks(assignee_id);
```

```
-- AUTO-UPDATE updated_at TRIGGER
```

```

CREATE OR REPLACE FUNCTION set_updated_at()
RETURNS TRIGGER AS $$
BEGIN
  NEW.updated_at = NOW();
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_organizations_updated_at
  BEFORE UPDATE ON organizations
  FOR EACH ROW EXECUTE FUNCTION set_updated_at();

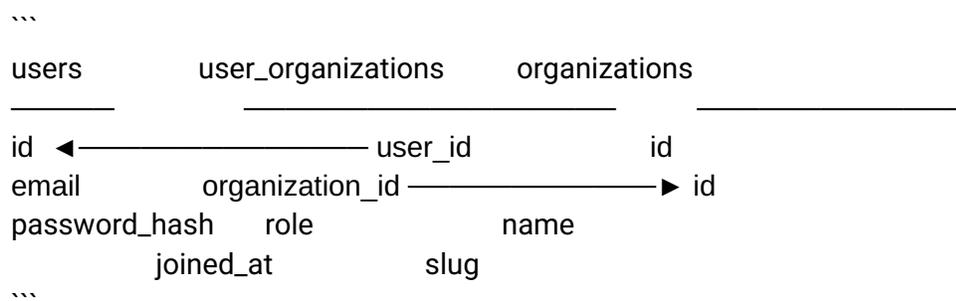
CREATE TRIGGER trg_users_updated_at
  BEFORE UPDATE ON users
  FOR EACH ROW EXECUTE FUNCTION set_updated_at();

CREATE TRIGGER trg_tasks_updated_at
  BEFORE UPDATE ON tasks
  FOR EACH ROW EXECUTE FUNCTION set_updated_at();
...

```

Relationship Breakdown

User ↔ Organization (Many-to-Many via `UserOrganization`)



****Why a join table instead of a simple foreign key:****

A user belongs to multiple organizations, each with a different role. That role lives on the *relationship*, not on the user or the org. A join table is the only correct model for this.

****`onDelete: Cascade` on both sides**** means:

- Delete a user - their memberships are removed


```
All tasks in an org          idx_tasks_org_id
Tasks in an org filtered by status      idx_tasks_org_status  -
composite
Tasks assigned to a user      idx_tasks_assignee_id
All members of an org        idx_user_org_org_id
All orgs a user belongs to    idx_user_org_user_id
...

```

The composite index `(organization_id, status)` covers the single-column `organization_id` queries too – PostgreSQL can use the leftmost prefix. This means `idx_tasks_org_id` as a standalone index is redundant if you have the composite, but it is kept explicit here for clarity and because query planners sometimes prefer it for simple scans.

Migration Command

```
```bash
npx prisma migrate dev --name add-organizations-users-tasks
...

```

**USER: I want to use Prisma as the ORM. Convert that schema to a Prisma schema file and explain any adjustments you'd make for best practices.**

## # Prisma Schema – Conversion & Best Practices

The schema from the previous step was already written in Prisma format. This response focuses on the **adjustments and best practices** applied on top of that baseline, with clear explanations for each decision.

---

## ## Final `prisma/schema.prisma`

```
```prisma
generator client {
  provider = "prisma-client-js"
}

```

```
datasource db {  
  provider = "postgresql"  
  url      = env("DATABASE_URL")  
}
```

```
// _____  
// ENUMS  
// _____
```

```
enum OrgRole {  
  OWNER  
  ADMIN  
  MEMBER  
}
```

```
enum TaskStatus {  
  TODO  
  IN_PROGRESS  
  DONE  
}
```

```
enum TaskPriority {  
  LOW  
  MEDIUM  
  HIGH  
}
```

```
// _____  
// MODELS  
// _____
```

```
model Organization {  
  id      String @id @default(cuid())  
  name    String @db.VarChar(255)  
  slug    String @unique @db.VarChar(100)  
  createdAt DateTime @default(now())  
  updatedAt DateTime @updatedAt
```

```
  memberships UserOrganization[]  
  tasks       Task[]
```

```
    @@map("organizations")
}
```

```
model User {
  id      String @id @default(cuid())
  email   String @unique @db.VarChar(255)
  passwordHash String @map("password_hash")
  firstName String @map("first_name") @db.VarChar(100)
  lastName String @map("last_name") @db.VarChar(100)
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt

  memberships UserOrganization[]
  assignedTasks Task[] @relation("TaskAssignee")
  createdTasks Task[] @relation("TaskCreator")
}
```

```
    @@map("users")
}
```

```
model UserOrganization {
  id      String @id @default(cuid())
  role    OrgRole @default(MEMBER)
  joinedAt DateTime @default(now())

  userId String @map("user_id")
  user    User @relation(fields: [userId], references: [id], onDelete: Cascade)

  organizationId String @map("organization_id")
  organization    Organization @relation(fields: [organizationId], references: [id], onDelete: Cascade)

  @@unique([userId, organizationId])
  @@index([organizationId])
  @@index([userId])
  @@map("user_organizations")
}
```

```
model Task {
  id      String @id @default(cuid())
  title   String @db.VarChar(500)
  description String? @db.Text
  status  TaskStatus @default(TODO)
}
```

```
priority TaskPriority @default(MEDIUM)
dueDate DateTime? @map("due_date")
createdAt DateTime @default(now())
updatedAt DateTime @updatedAt
```

```
organizationId String @map("organization_id")
organization Organization @relation(fields: [organizationId], references: [id], onDelete:
Cascade)
```

```
creatorId String @map("creator_id")
creator User @relation("TaskCreator", fields: [creatorId], references: [id], onDelete: Restrict)
```

```
assigneeId String? @map("assignee_id")
assignee User? @relation("TaskAssignee", fields: [assigneeId], references: [id], onDelete:
SetNull)
```

```
@@index([organizationId])
@@index([organizationId, status])
@@index([assigneeId])
@@map("tasks")
}
```

```
...
```

```
---
```

Adjustments Made and Why

1. `@@map()` and `@map()` – Snake Case in the Database

```
``prisma
// Prisma model name (PascalCase, used in TypeScript)
model UserOrganization {
  passwordHash String @map("password_hash") // column name in PostgreSQL

  @@map("user_organizations") // table name in PostgreSQL
}
...

```

****Why:**** PostgreSQL convention is `snake_case`. TypeScript convention is `camelCase`. Without `@map`, Prisma uses the TypeScript name as the column name, producing columns like `passwordHash` and `firstName` in your database – which is non-standard and makes raw SQL queries painful.

With `@map`/`@@map``, you get:

- TypeScript: `user.passwordHash``, `user.firstName`` (camelCase)
- PostgreSQL: `password_hash``, `first_name`` (snake_case)

Both sides follow their own conventions.

2. `@db.VarChar()`` and `@db.Text`` – Explicit Column Types

```
``prisma
name    String @db.VarChar(255)
slug    String @db.VarChar(100)
description String? @db.Text
``
```

****Why:**** Without `@db`` annotations, Prisma maps every `String`` to `TEXT`` in PostgreSQL. That is technically fine, but:

- `VarChar(n)`` communicates intent and enforces a length constraint at the DB level
- `TEXT`` is appropriate for unbounded content like `description``
- It makes the migration SQL match what a DBA would write by hand

Field	Type	Reason
<code>email`</code>	<code>VarChar(255)`</code>	RFC 5321 max email length
<code>name`</code> , <code>title`</code>	<code>VarChar(255/500)`</code>	Bounded, searchable strings
<code>slug`</code>	<code>VarChar(100)`</code>	Short by design
<code>description`</code>	<code>Text`</code>	Unbounded free text

3. `@updatedAt`` – Let Prisma Manage Timestamps

```
``prisma
updatedAt DateTime @updatedAt
``
```

****Why:**** In the raw SQL schema, a trigger was needed to auto-update `updated_at``. Prisma's `@updatedAt`` handles this automatically on every `update`` operation – no trigger required. This removes a piece of infrastructure that Prisma makes unnecessary.

****Important caveat:**** `@updatedAt` only fires when you use Prisma Client. If you run raw SQL updates directly against the database, the column will not auto-update. If you need raw SQL writes, keep the trigger.

4. Named Relations – Required, Not Optional

```
``prisma
model User {
  assignedTasks Task[] @relation("TaskAssignee")
  createdTasks Task[] @relation("TaskCreator")
}

model Task {
  creator User @relation("TaskCreator", ...)
  assignee User? @relation("TaskAssignee", ...)
}
...

```

****Why:**** When two relations point to the same model, Prisma cannot infer which FK maps to which relation. Named relations are required here – Prisma will throw a schema validation error without them. The names are arbitrary strings; they just need to match on both sides.

5. `onDelete` Behavior – Explicit on Every FK

Every foreign key has an explicit `onDelete` rule:

```
``prisma
// UserOrganization
user      @relation(..., onDelete: Cascade) // delete user → remove
memberships
organization @relation(..., onDelete: Cascade) // delete org → remove
memberships

// Task
organization @relation(..., onDelete: Cascade) // delete org → delete
its tasks

```

```
creator    @relation(..., onDelete: Restrict) // block user deletion if they own tasks
assignee   @relation(..., onDelete: SetNull) // unassign tasks if user is deleted
...

```

****Why:**** Prisma's default `onDelete` is `SetNull` for optional relations and `Restrict` for required ones – but relying on defaults is a maintenance hazard. When someone reads the schema six months from now, explicit rules make the data integrity contract obvious without requiring knowledge of Prisma defaults.

```
| Rule | Effect |
|--|--|
| `Cascade` | Child rows deleted when parent is deleted |
| `Restrict` | Parent deletion blocked if child rows exist |
| `SetNull` | FK column set to NULL when parent is deleted |

```

6. `cuid()` over `uuid()` for IDs

```
```prisma
id String @id @default(cuid())
...

```

**\*\*Why:\*\*** Both are valid. `cuid()` is chosen here because:

- CUIDs are URL-safe by default (no hyphens to encode)
- They are monotonically increasing, which reduces B-tree index fragmentation compared to random UUIDs
- They are generated in application code, not the database, keeping ID generation consistent regardless of DB

If you prefer `uuid()`, use `@default(uuid())` – it is a valid alternative with no functional difference for this use case.

---

### ## Running the Migration

```
```bash
# Generate the migration SQL and apply it
npx prisma migrate dev --name init

# Regenerate the Prisma Client after any schema change

```

```
npx prisma generate
```

```
# Inspect your database visually
```

```
npx prisma studio
```

```
...
```

```
---
```

```
## What the Generated Client Looks Like
```

After `prisma generate`, the type-safe client reflects every schema decision:

```
``typescript
```

```
import prisma from "./config/database";
```

```
// Fully typed – TypeScript knows organizationId is required,
```

```
// assigneeId is optional, status is TaskStatus enum
```

```
const task = await prisma.task.create({
```

```
  data: {
```

```
    title: "Write tests",
```

```
    organizationId: "org_123",
```

```
    creatorId: "user_456",
```

```
    status: "TODO",
```

```
    priority: "HIGH",
```

```
  },
```

```
});
```

```
// Relation traversal is type-safe
```

```
const org = await prisma.organization.findUnique({
```

```
  where: { slug: "acme-corp" },
```

```
  include: {
```

```
    memberships: {
```

```
      include: { user: true },
```

```
    },
```

```
    tasks: {
```

```
      where: { status: "IN_PROGRESS" },
```

```
    },
```

```
  },
```

```
});
```

```
...
```

USER: Set up the authentication middleware. Users should authenticate with JWT tokens. Show me the middleware implementation and explain how it should integrate with our Express routes.

Authentication Middleware Implementation

Files to Create/Modify

...

```
src/
├── middleware/
│   ├── authenticate.ts    # JWT verification middleware
│   └── requireRole.ts    # Role-based access control middleware
├── modules/auth/
│   ├── auth.routes.ts
│   ├── auth.controller.ts
│   └── auth.service.ts
└── types/index.ts        # Update with new types
```

...

Updated `src/types/index.ts`

``typescript

```
import { Request } from "express";
import { OrgRole } from "@prisma/client";

export interface JwtPayload {
  sub: string;    // userId – "sub" is the JWT standard claim for subject
  email: string;
  iat?: number;  // issued at – set automatically by jsonwebtoken
  exp?: number;  // expiry – set automatically by jsonwebtoken
}
```

```
// Attached by authenticate middleware
export interface AuthenticatedRequest extends Request {
  user: JwtPayload;
}
```

```
// Attached by requireRole middleware (used on org-scoped routes)
export interface OrgScopedRequest extends AuthenticatedRequest {
  membership: {
    organizationId: string;
    role: OrgRole;
  };
}
...

```

****Why `sub` instead of `userId`:** `sub` (subject) is the IANA-registered JWT claim for identifying the principal. Using standard claim names means any JWT library or debugging tool understands the token structure without custom documentation.

```
## `src/middleware/authenticate.ts`
```

```
``typescript
```

```
import { Response, NextFunction } from "express";
import jwt from "jsonwebtoken";
import { env } from "../config/env";
import { AppError } from "../utils/AppError";
import { AuthenticatedRequest, JwtPayload } from "../types";
```

```
export function authenticate(
  req: AuthenticatedRequest,
  res: Response,
  next: NextFunction
): void {
  // 1. Extract token from Authorization header
  const authHeader = req.headers.authorization;

  if (!authHeader || !authHeader.startsWith("Bearer ")) {
    return next(new AppError("Authorization header missing or malformed", 401));
  }

  const token = authHeader.split(" ")[1];

  if (!token) {
    return next(new AppError("Token not provided", 401));
  }
}
```

```

// 2. Verify signature and expiry
try {
  const decoded = jwt.verify(token, env.JWT_SECRET) as JwtPayload;
  req.user = decoded;
  next();
} catch (err) {
  if (err instanceof jwt.TokenExpiredError) {
    return next(new AppError("Token has expired", 401));
  }
  if (err instanceof jwt.JsonWebTokenError) {
    return next(new AppError("Invalid token", 401));
  }
  // Unexpected error – let the global handler deal with it
  next(err);
}
}
...

```

****Why distinguish `TokenExpiredError` from `JsonWebTokenError`:**** Clients need to know *why* a 401 happened. An expired token means "refresh and retry." An invalid token means "re-authenticate." Returning the same message for both forces clients to guess.

```

### `src/middleware/requireRole.ts`

```

```

``typescript

```

```

import { Response, NextFunction } from "express";
import { OrgRole } from "@prisma/client";
import prisma from "../config/database";
import { AppError } from "../utils/AppError";
import { AuthenticatedRequest, OrgScopedRequest } from "../types";

// Role hierarchy – higher index = more permissions
const ROLE_HIERARCHY: OrgRole[] = ["MEMBER", "ADMIN", "OWNER"];

function hasRequiredRole(userRole: OrgRole, requiredRole: OrgRole): boolean {
  return ROLE_HIERARCHY.indexOf(userRole) >= ROLE_HIERARCHY.indexOf(requiredRole);
}

// Factory function – returns middleware configured for a specific org param and minimum role
export function requireRole(minimumRole: OrgRole = "MEMBER") {

```

```

return async (
  req: AuthenticatedRequest,
  res: Response,
  next: NextFunction
): Promisevoid = {
  const { organizationId } = req.params;

  if (!organizationId) {
    return next(new AppError("Organization ID missing from route params", 400));
  }

  // Verify the authenticated user is actually a member of this org
  const membership = await prisma.userOrganization.findUnique({
    where: {
      userId_organizationId: {
        userId: req.user.sub,
        organizationId,
      },
    },
    select: { role: true, organizationId: true },
  });

  if (!membership) {
    // Intentionally vague – don't reveal whether the org exists
    return next(new AppError("Organization not found or access denied", 404));
  }

  if (!hasRequiredRole(membership.role, minimumRole)) {
    return next(
      new AppError(
        `This action requires ${minimumRole} role or higher`,
        403
      )
    );
  }

  // Attach membership to request so downstream handlers don't re-query
  (req as OrgScopedRequest).membership = {
    organizationId: membership.organizationId,
    role: membership.role,
  };
};

```

```
    next();
  };
}
...
```

****Why a factory function:**** `requireRole("ADMIN")` reads clearly at the route definition level. It also means the same middleware handles all role levels – no separate `requireAdmin`, `requireOwner` functions to maintain.

****Why 404 instead of 403 when membership is missing:**** Returning 403 confirms the organization exists. Returning 404 reveals nothing about whether the org exists at all. This is the standard approach for multi-tenant APIs to prevent org enumeration.

```
## `src/modules/auth/auth.service.ts`
```

```
``typescript
```

```
import bcrypt from "bcryptjs";
import jwt from "jsonwebtoken";
import prisma from "../../config/database";
import { env } from "../../config/env";
import { AppError } from "../../utils/AppError";
import { JwtPayload } from "../../types";
```

```
const SALT_ROUNDS = 12;
```

```
export const authService = {
  async register(email: string, password: string, firstName: string, lastName: string) {
    const existing = await prisma.user.findUnique({ where: { email } });
    if (existing) {
      // Same message for "email taken" and "user exists" – prevents user enumeration
      throw new AppError("Invalid credentials or email already in use", 409);
    }
  }
}
```

```
const passwordHash = await bcrypt.hash(password, SALT_ROUNDS);
```

```
const user = await prisma.user.create({
  data: { email, passwordHash, firstName, lastName },
  select: {
    id: true,
    email: true,
```

```

    firstName: true,
    lastName: true,
    createdAt: true,
    // passwordHash intentionally excluded from return value
  },
});

return user;
},

async login(email: string, password: string) {
  const user = await prisma.user.findUnique({ where: { email } });

  // Always run bcrypt.compare even if user not found
  // This prevents timing attacks that reveal whether an email is registered
  const passwordHash = user?.passwordHash ??
"$2b$12$invalidhashpadding0000000000000000000000000000000000000000";
  const isValid = await bcrypt.compare(password, passwordHash);

  if (!user || !isValid) {
    throw new AppError("Invalid email or password", 401);
  }

  const token = authService.signToken({ sub: user.id, email: user.email });

  return {
    token,
    user: {
      id: user.id,
      email: user.email,
      firstName: user.firstName,
      lastName: user.lastName,
    },
  };
},

signToken(payload: JwtPayload): string {
  return jwt.sign(payload, env.JWT_SECRET, {
    expiresIn: env.JWT_EXPIRES_IN,
  });
},
};

```

...

****Why always run `bcrypt.compare`:**** If you return early when a user isn't found, the response time is measurably faster than when bcrypt runs. An attacker can use response timing to enumerate valid email addresses. Running bcrypt against a dummy hash makes both paths take the same time.

```
### `src/modules/auth/auth.controller.ts`
```

```
``typescript
```

```
import { Request, Response, NextFunction } from "express";
import { z } from "zod";
import { authService } from "../auth.service";
import { AppError } from "../../utils/AppError";
```

```
const registerSchema = z.object({
  email: z.string().email(),
  password: z.string().min(8, "Password must be at least 8 characters"),
  firstName: z.string().min(1).max(100),
  lastName: z.string().min(1).max(100),
});
```

```
const loginSchema = z.object({
  email: z.string().email(),
  password: z.string().min(1),
});
```

```
export const authController = {
  async register(req: Request, res: Response, next: NextFunction) {
    try {
      const result = registerSchema.safeParse(req.body);
      if (!result.success) {
        throw new AppError(result.error.errors[0].message, 400);
      }

      const { email, password, firstName, lastName } = result.data;
      const user = await authService.register(email, password, firstName, lastName);

      res.status(201).json({ status: "success", data: { user } });
    } catch (err) {
```

```
    next(err);
  }
},

async login(req: Request, res: Response, next: NextFunction) {
  try {
    const result = loginSchema.safeParse(req.body);
    if (!result.success) {
      throw new AppError("Invalid email or password format", 400);
    }

    const { email, password } = result.data;
    const { token, user } = await authService.login(email, password);

    res.status(200).json({ status: "success", data: { token, user } });
  } catch (err) {
    next(err);
  }
},
};
...

```

```
---

## `src/modules/auth/auth.routes.ts`

```

```
``typescript
import { Router } from "express";
import { authController } from "../auth.controller";

const router = Router();

router.post("/register", authController.register);
router.post("/login", authController.login);

export default router;
...

```

```
---

## Updated `src/app.ts`

```

```

``typescript
import express from "express";
import { env } from "../config/env";
import { errorHandler } from "../middleware/errorHandler";
import authRoutes from "../modules/auth/auth.routes";

const app = express();

app.use(express.json());

app.get("/health", (req, res) = {
  res.json({ status: "ok", environment: env.NODE_ENV });
});

// Public routes – no authentication required
app.use("/api/v1/auth", authRoutes);

// Protected routes will be mounted here in subsequent steps:
// app.use("/api/v1/organizations/:organizationId/tasks", authenticate, requireRole("MEMBER"),
// taskRoutes);

app.use(errorHandler);

app.listen(env.PORT, () = {
  console.log(`🚀 Server running on port ${env.PORT} [${env.NODE_ENV}]`);
});

export default app;

```

How the Middleware Chain Works

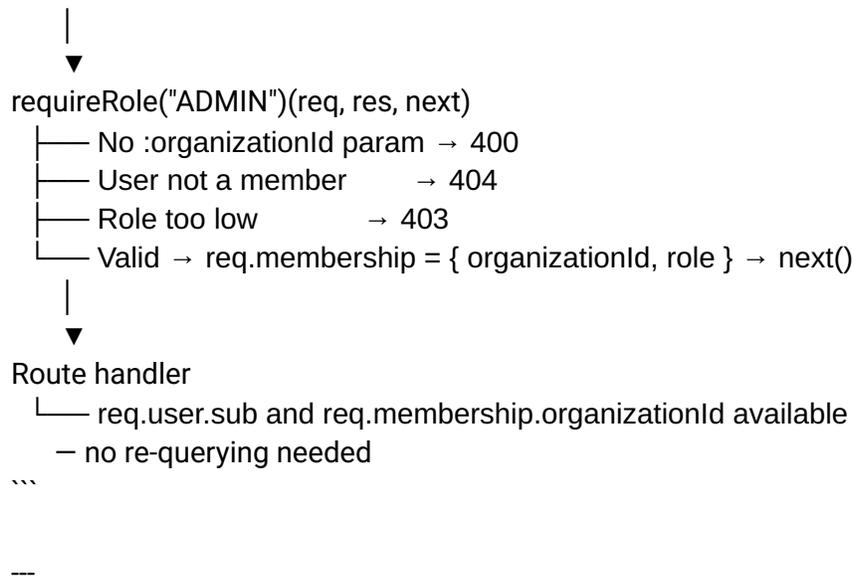
...

Incoming request



authenticate(req, res, next)

- ├── No/malformed header → 401
- ├── Expired token → 401 "Token has expired"
- ├── Invalid signature → 401 "Invalid token"
- └── Valid → req.user = { sub, email } → next()



Route-Level Usage Pattern

When task routes are added, this is how the middleware composes:

```

```typescript
// Every task route requires authentication + org membership
router.get(
 "/api/v1/organizations/:organizationId/tasks",
 authenticate,
 requireRole("MEMBER"), // any member can list tasks
 taskController.list
);

router.delete(
 "/api/v1/organizations/:organizationId/tasks/:taskId",
 authenticate,
 requireRole("ADMIN"), // only admins and owners can delete
 taskController.delete
);
```
```

The ``organizationId`` in the route param is what ``requireRole`` uses to look up the membership. This means tenant isolation is enforced at the middleware layer — a task controller never needs to verify org membership itself.

Optional / Future Enhancements

- **Refresh tokens** – issue short-lived access tokens (15 min) and long-lived refresh tokens stored in the database, enabling token revocation
- **Token blacklist** – store revoked JWTs in Redis until their natural expiry to support logout invalidation
- **Rate limiting on `/auth/login`** – prevent brute-force attacks with `express-rate-limit` on the login endpoint specifically

USER: Create a base error handling system. I want consistent error responses across the API with proper HTTP status codes. Show me the error classes and error handling middleware.

Error Handling System

Files to Create/Modify

```
...
src/
├── errors/
│   ├── AppError.ts      # Replace src/Utils/AppError.ts
│   ├── HttpErrors.ts   # Named error subclasses
│   └── PrismaErrors.ts  # Prisma-specific error mapping
├── middleware/
│   └── errorHandler.ts  # Update existing file
└── types/index.ts      # Add error response type
...
```

Remove `src/Utils/AppError.ts` – it is replaced by `src/errors/AppError.ts`.

`src/errors/AppError.ts`

```
``typescript
export class AppError extends Error {
  public readonly statusCode: number;
  public readonly isOperational: boolean;
  public readonly code: string;

  constructor(message: string, statusCode: number, code: string, isOperational = true) {
```

```

super(message);
this.statusCode = statusCode;
this.isOperational = isOperational;
this.code = code;

// Restores the correct prototype chain after extending a built-in class
Object.setPrototypeOf(this, new.target.prototype);

// Removes AppError constructor frames from the stack trace
Error.captureStackTrace(this, this.constructor);
}
}
...

```

****Why `isOperational`:**** Errors split into two categories:

| Type | `isOperational` | Meaning | Response |
|-------------|-----------------|---|----------------------------------|
| Operational | `true` | Expected failure – bad input, not found, unauthorized | Return structured JSON to client |
| Programmer | `false` | Bug – null reference, type error, logic fault | Log fully, return generic 500 |

The error handler uses this flag to decide how much detail to expose.

****Why `code`:**** HTTP status codes tell clients *how bad* something is. Error codes tell them *what happened* so they can react programmatically – show the right UI message, retry, redirect to login, etc.

```
## `src/errors/HttpErrors.ts`
```

```

``typescript
import { AppError } from "../AppError";

export class BadRequestError extends AppError {
  constructor(message = "Bad request", code = "BAD_REQUEST") {
    super(message, 400, code);
  }
}

export class UnauthorizedError extends AppError {

```

```
    constructor(message = "Authentication required", code = "UNAUTHORIZED") {
      super(message, 401, code);
    }
  }
}
```

```
export class ForbiddenError extends AppError {
  constructor(message = "Access denied", code = "FORBIDDEN") {
    super(message, 403, code);
  }
}
```

```
export class NotFoundError extends AppError {
  constructor(resource = "Resource", code = "NOT_FOUND") {
    super(`${resource} not found`, 404, code);
  }
}
```

```
export class ConflictError extends AppError {
  constructor(message = "Resource already exists", code = "CONFLICT") {
    super(message, 409, code);
  }
}
```

```
export class UnprocessableEntityError extends AppError {
  constructor(message = "Validation failed", code = "VALIDATION_ERROR") {
    super(message, 422, code);
  }
}
```

```
export class InternalServerError extends AppError {
  constructor(message = "Internal server error", code = "INTERNAL_ERROR") {
    // isOperational = false – this is always a bug, never expected
    super(message, 500, code, false);
  }
}
...

```

****Why subclasses instead of passing status codes directly:****

```
```typescript
// Without subclasses – magic numbers, easy to mistype
throw new AppError("Not found", 404, "NOT_FOUND");

```

```
// With subclasses – intent is clear, status code is encapsulated
throw new NotFoundError("Task");
````
```

Subclasses also enable `instanceof` checks in tests and in the error handler without inspecting status codes.

```
## `src/errors/PrismaErrors.ts`
```

```
``typescript
```

```
import { Prisma } from "@prisma/client";
```

```
import { AppError } from "../AppError";
```

```
import { ConflictError, NotFoundError, InternalServerError } from "../HttpErrors";
```

```
// Prisma error codes reference:
```

```
// https://www.prisma.io/docs/reference/api-reference/error-reference
```

```
export function mapPrismaError(err: unknown): AppError {
  if (err instanceof Prisma.PrismaClientKnownRequestError) {
    switch (err.code) {
      // Unique constraint violation
      case "P2002": {
        const fields = (err.meta?.target as string[])?.join(", ") ?? "field";
        return new ConflictError(`A record with this ${fields} already exists`, "DUPLICATE_ENTRY");
      }

      // Record not found (findUniqueOrThrow / findFirstOrThrow)
      case "P2025":
        return new NotFoundError("Record", "RECORD_NOT_FOUND");

      // Foreign key constraint violation
      case "P2003": {
        const field = (err.meta?.field_name as string) ?? "reference";
        return new AppError(`Related ${field} does not exist`, 400, "INVALID_REFERENCE");
      }

      // Operation on a record that does not exist
      case "P2016":
        return new NotFoundError("Record", "RECORD_NOT_FOUND");
    }
  }
}
```

```

// Null constraint violation
case "P2011": {
  const field = (err.meta?.constraint as string) ?? "field";
  return new AppError(` ${field} cannot be null`, 400, "NULL_CONSTRAINT");
}

default:
  return new InternalServerError();
}
}

if (err instanceof Prisma.PrismaClientValidationError) {
  // Malformed query – always a programmer error
  return new InternalServerError();
}

if (err instanceof Prisma.PrismaClientInitializationError) {
  // DB connection failed – not the client's fault
  return new InternalServerError();
}

return new InternalServerError();
}
...

```

****Why a dedicated Prisma mapper:**** Prisma errors are not `AppError` instances. Without this mapper, raw Prisma errors would fall through to the generic 500 handler even when they represent known, handleable conditions like unique constraint violations. The mapper converts them into the correct operational error type.

```
## Updated `src/middleware/errorHandler.ts`
```

```

``typescript
import { Request, Response, NextFunction } from "express";
import { ZodError } from "zod";
import { Prisma } from "@prisma/client";
import { AppError } from "../errors/AppError";
import { mapPrismaError } from "../errors/PrismaErrors";
import { env } from "../config/env";

```

```
interface ErrorResponse {
  status: "error";
  code: string;
  message: string;
  errors?: Recordstring, string[]; // Field-level validation errors
  stack?: string; // Development only
}

export function errorHandler(
  err: unknown,
  req: Request,
  res: Response,
  next: NextFunction // Required 4th param – Express uses arity to detect error handlers
): void {
  // — 1. Normalize to AppError
```

```
  let appError: AppError;

  if (err instanceof AppError) {
    appError = err;
  } else if (err instanceof ZodError) {
    appError = normalizeZodError(err);
  } else if (isPrismaError(err)) {
    appError = mapPrismaError(err);
  } else if (err instanceof SyntaxError && "body" in err) {
    // express.json() parse failure
    appError = new AppError("Invalid JSON in request body", 400, "INVALID_JSON");
  } else {
    // Unknown error – treat as non-operational
    appError = new AppError("Internal server error", 500, "INTERNAL_ERROR", false);
  }
```

```
  // — 2. Log
```

```
  if (!appError.isOperational) {
    // Non-operational errors are bugs – log the full error
    console.error({
      type: "UNHANDLED_ERROR",
      message: appError.message,
```

```
    stack: appError.stack,
    url: req.url,
    method: req.method,
  });
}
```

```
// — 3. Build response
```

```
const body: ErrorResponse = {
  status: "error",
  code: appError.code,
  message: appError.isOperational
    ? appError.message
    : "An unexpected error occurred", // Never expose bug details to clients
};
```

```
// Attach field errors if present (set by normalizeZodError)
if ((appError as AppError & { errors?: Recordstring, string[] }).errors) {
  body.errors = (appError as AppError & { errors?: Recordstring, string[] }).errors;
}
```

```
// Stack trace in development only
if (env.NODE_ENV === "development") {
  body.stack = appError.stack;
}
```

```
res.status(appError.statusCode).json(body);
}
```

```
// — Helpers
```

```
function normalizeZodError(err: ZodError): AppError {
  // Group Zod issues by field path
  const errors: Recordstring, string[] = {};

  for (const issue of err.issues) {
    const path = issue.path.join(".") || "root";
    if (!errors[path]) errors[path] = [];
    errors[path].push(issue.message);
  }
}
```

```

const appError = new AppError(
  "Validation failed",
  422,
  "VALIDATION_ERROR"
) as AppError & { errors: Recordstring, string[] };

appError.errors = errors;
return appError;
}

```

```

function isPrismaError(err: unknown): boolean {
  return (
    err instanceof Prisma.PrismaClientKnownRequestError ||
    err instanceof Prisma.PrismaClientValidationError ||
    err instanceof Prisma.PrismaClientInitializationError
  );
}
...

```

Update Import Paths

The previous `authenticate.ts` and `requireRole.ts` import from `../utils/AppError`. Update those imports:

```

```typescript
// Before
import { AppError } from "../utils/AppError";

// After
import { AppError } from "../errors/AppError";
import { UnauthorizedError, ForbiddenError, NotFoundError } from "../errors/HttpErrors";
...

```

Update the middleware to use the named subclasses:

```

```typescript
// src/middleware/authenticate.ts – replace throw statements

// Before

```

```

return next(new AppError("Authorization header missing or malformed", 401));

// After
return next(new UnauthorizedError("Authorization header missing or malformed",
"MISSING_TOKEN"));

// Before
return next(new AppError("Token has expired", 401));

// After
return next(new UnauthorizedError("Token has expired", "TOKEN_EXPIRED"));

// Before
return next(new AppError("Invalid token", 401));

// After
return next(new UnauthorizedError("Invalid token", "TOKEN_INVALID"));
...

``typescript
// src/middleware/requireRole.ts – replace throw statements

return next(new NotFoundError("Organization", "ORG_NOT_FOUND"));
return next(new ForbiddenError("This action requires ${minimumRole} role or higher",
"INSUFFICIENT_ROLE"));
...

---

## Response Shape Reference

```

Every error from the API now has a consistent shape:

```

``typescript
// 401 – Expired token
{
  "status": "error",
  "code": "TOKEN_EXPIRED",
  "message": "Token has expired"
}

// 422 – Validation failure (Zod)

```

```
{
  "status": "error",
  "code": "VALIDATION_ERROR",
  "message": "Validation failed",
  "errors": {
    "email": ["Invalid email"],
    "password": ["Password must be at least 8 characters"]
  }
}
```

// 409 – Unique constraint from Prisma

```
{
  "status": "error",
  "code": "DUPLICATE_ENTRY",
  "message": "A record with this email already exists"
}
```

// 500 – Unhandled bug (production)

```
{
  "status": "error",
  "code": "INTERNAL_ERROR",
  "message": "An unexpected error occurred"
}
```

// 500 – Unhandled bug (development only – stack included)

```
{
  "status": "error",
  "code": "INTERNAL_ERROR",
  "message": "An unexpected error occurred",
  "stack": "Error: ..."
}
...

```

Error Flow Summary

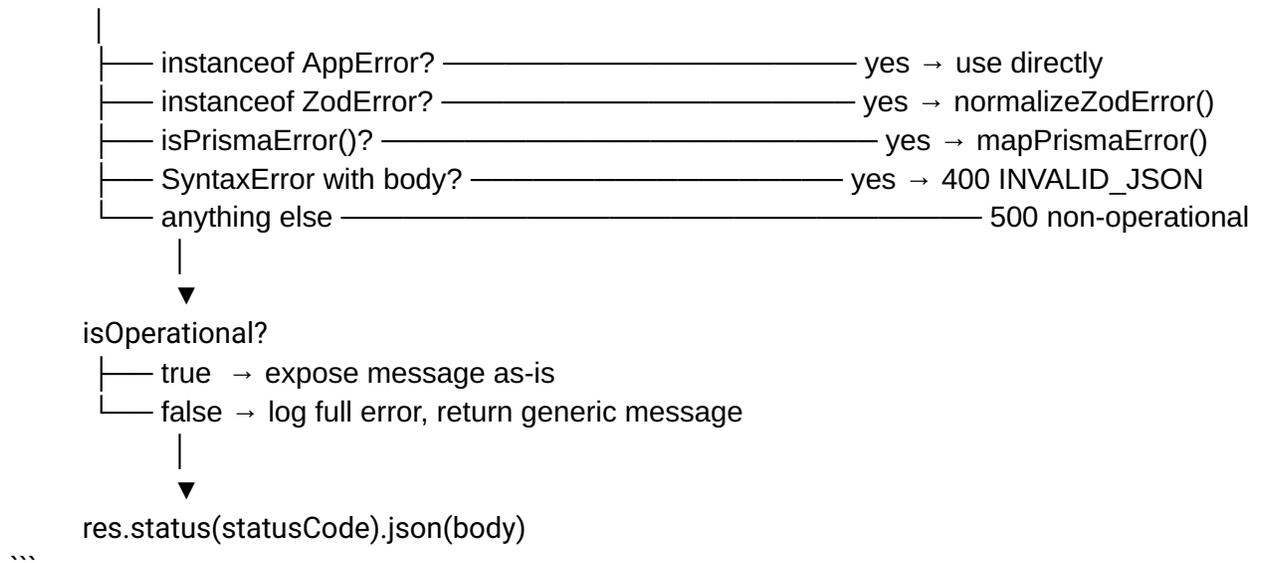
...

```
throw new NotFoundError("Task")
```

|



```
errorHandler(err, req, res, next)
```



Optional / Future Enhancements

- **Structured logging** – replace `console.error` with a logger like `pino` or `winston` to emit JSON log lines with request IDs, enabling log aggregation in Datadog, CloudWatch, etc.
- **Request ID tracing** – attach a `X-Request-ID` header on every request and include it in error responses and logs so a client-reported error code can be traced to a specific log entry
- **Error monitoring** – integrate Sentry (`@sentry/node`) in the non-operational branch to capture stack traces and alert on new error types automatically

USER: Build the user registration endpoint. It should:

- **Validate email format**
- **Hash passwords**
- **Create the user in the database**
- **Return a JWT token**

Use the error handling system we set up earlier.

Note: Specific factual details should be independently verified. If uncertainty exists, consult authoritative sources.

User Registration Endpoint

Files to Create/Modify

...

src/

```
|— modules/auth/
|   |— auth.routes.ts    # Already exists — no changes needed
|   |— auth.controller.ts # Update existing file
|   |— auth.service.ts   # Update existing file
|   |— auth.validation.ts # New — extract validation schemas
|   ...
|
|
|
|
```

```
---

## `src/modules/auth/auth.validation.ts`
```

```
``typescript
```

```
import { z } from "zod";
```

```
export const registerSchema = z.object({
```

```
  email: z
```

```
    .string({ required_error: "Email is required" })
```

```
    .email("Invalid email format")
```

```
    .toLowerCase() // Normalize before storing — prevents "User@example.com" vs
```

```
"user@example.com" duplicates
```

```
    .max(255, "Email must not exceed 255 characters"),
```

```
  password: z
```

```
    .string({ required_error: "Password is required" })
```

```
    .min(8, "Password must be at least 8 characters")
```

```
    .max(72, "Password must not exceed 72 characters") // bcrypt silently truncates at 72 bytes
```

```
    .regex(/[A-Z]/, "Password must contain at least one uppercase letter")
```

```
    .regex(/[0-9]/, "Password must contain at least one number"),
```

```
  firstName: z
```

```
    .string({ required_error: "First name is required" })
```

```
    .trim()
```

```
    .min(1, "First name cannot be empty")
```

```
    .max(100, "First name must not exceed 100 characters"),
```

```
  lastName: z
```

```
    .string({ required_error: "Last name is required" })
```

```
    .trim()
```

```
    .min(1, "Last name cannot be empty")
```

```
    .max(100, "Last name must not exceed 100 characters"),
```

```
});
```

```
export type RegisterInput = z.inferTypeof registerSchema;
```

...

****Why `.toLowerCase()` on email:**** Email addresses are case-insensitive per RFC 5321. Normalizing at the validation layer means the database always stores lowercase, so `@unique` on `email` works correctly and lookups never need `LOWER()` calls.

****Why max 72 on password:**** `bcrypt` processes at most 72 bytes. A password of 73+ characters is silently truncated to 72 before hashing. Without this limit, two different passwords that share the same first 72 characters would produce the same hash — a subtle security flaw.

```
## `src/modules/auth/auth.service.ts`
```

```
``typescript
```

```
import bcrypt from "bcryptjs";
import jwt from "jsonwebtoken";
import prisma from "../../config/database";
import { env } from "../../config/env";
import { ConflictError } from "../../errors/HttpErrors";
import { JwtPayload } from "../../types";
import { RegisterInput } from "./auth.validation";
```

```
const SALT_ROUNDS = 12;
```

```
export const authService = {
  async register(input: RegisterInput) {
    // — 1. Check for existing user
```

```
const existing = await prisma.user.findUnique({
  where: { email: input.email },
  select: { id: true }, // Minimal select — we only need to know if it exists
});
```

```
if (existing) {
  throw new ConflictError(
    "An account with this email already exists",
    "EMAIL_TAKEN"
  );
}
```

```
// — 2. Hash password
```

```
const passwordHash = await bcrypt.hash(input.password, SALT_ROUNDS);
```

```
// — 3. Create user
```

```
const user = await prisma.user.create({
  data: {
    email: input.email,
    passwordHash,
    firstName: input.firstName,
    lastName: input.lastName,
  },
  select: {
    id: true,
    email: true,
    firstName: true,
    lastName: true,
    createdAt: true,
    // passwordHash deliberately excluded — never returned to callers
  },
});
```

```
// — 4. Issue token
```

```
const token = authService.signToken({
  sub: user.id,
  email: user.email,
});

return { user, token };
},

async login(email: string, password: string) {
  const user = await prisma.user.findUnique({ where: { email } });

  // Always run bcrypt even when user not found — prevents timing attacks
  const hashToCompare =
    user?.passwordHash ??
    "$2b$12$invalidhashpaddinginvalidhashpaddinginvalidhashpadding";
```

```

const isValid = await bcrypt.compare(password, hashToCompare);

if (!user || !isValid) {
  throw new ConflictError("Invalid email or password", "INVALID_CREDENTIALS");
}

const token = authService.signToken({ sub: user.id, email: user.email });

return {
  token,
  user: {
    id: user.id,
    email: user.email,
    firstName: user.firstName,
    lastName: user.lastName,
  },
};
},

signToken(payload: JwtPayload): string {
  return jwt.sign(payload, env.JWT_SECRET, {
    expiresIn: env.JWT_EXPIRES_IN,
  });
},
};
...

```

****Why `select` on `prisma.user.create`:**** Prisma returns all columns by default. Explicitly selecting fields ensures `passwordHash` is never accidentally included in a return value, even if the function is refactored later. This is a defense-in-depth measure – the hash should never leave the service layer.

****Why `SALT_ROUNDS = 12`:**** bcrypt cost factor is exponential. At 12, hashing takes ~250–400ms on typical hardware – slow enough to make brute-force attacks expensive, fast enough to not degrade API response times under normal load. The commonly cited minimum is 10; 12 is a reasonable current default.

```
## `src/modules/auth/auth.controller.ts`
```

```

``typescript
import { Request, Response, NextFunction } from "express";
import { registerSchema } from "../auth.validation";
import { authService } from "../auth.service";
import { AppError } from "../../errors/AppError";

export const authController = {
  async register(req: Request, res: Response, next: NextFunction): Promisevoid {
    try {
      // — Validate



---


      const result = registerSchema.safeParse(req.body);

      if (!result.success) {
        // Throw the raw ZodError — errorHandler normalizes it into field errors
        throw result.error;
      }

      // — Execute



---


      const { user, token } = await authService.register(result.data);

      // — Respond



---


      res.status(201).json({
        status: "success",
        data: { user, token },
      });
    } catch (err) {
      next(err);
    }
  },

  async login(req: Request, res: Response, next: NextFunction): Promisevoid {
    try {
      const { email, password } = req.body;

      if (typeof email !== "string" || typeof password !== "string") {
        throw new AppError("Email and password are required", 400, "BAD_REQUEST");
      }

      const { user, token } = await authService.login(
        email.toLowerCase().trim(),

```

```

        password
    );

    res.status(200).json({
        status: "success",
        data: { user, token },
    });
} catch (err) {
    next(err);
}
},
};
...

```

****Why `throw result.error` instead of `throw new AppError(...)`:**** The `errorHandler` middleware already handles `ZodError` instances and converts them into structured field-level error responses. Rethrowing the raw `ZodError` means the full field map is preserved – the controller does not need to manually format validation messages.

```
### `src/modules/auth/auth.routes.ts`
```

No changes needed from the previous step. Shown here for completeness:

```

``typescript
import { Router } from "express";
import { authController } from "../auth.controller";

const router = Router();

router.post("/register", authController.register);
router.post("/login", authController.login);

export default router;
...

```

```
## Request / Response Reference
```

```
### `POST /api/v1/auth/register`
```

****Request body:****

```
```json
{
 "email": "jane@example.com",
 "password": "SecurePass1",
 "firstName": "Jane",
 "lastName": "Doe"
}
```
```

****201 Created:****

```
```json
{
 "status": "success",
 "data": {
 "user": {
 "id": "clxyz123",
 "email": "jane@example.com",
 "firstName": "Jane",
 "lastName": "Doe",
 "createdAt": "2024-01-15T10:30:00.000Z"
 },
 "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."
 }
}
```
```

****409 Conflict – email already registered:****

```
```json
{
 "status": "error",
 "code": "EMAIL_TAKEN",
 "message": "An account with this email already exists"
}
```
```

****422 Unprocessable Entity – validation failure:****

```
```json
{
 "status": "error",
 "code": "VALIDATION_ERROR",

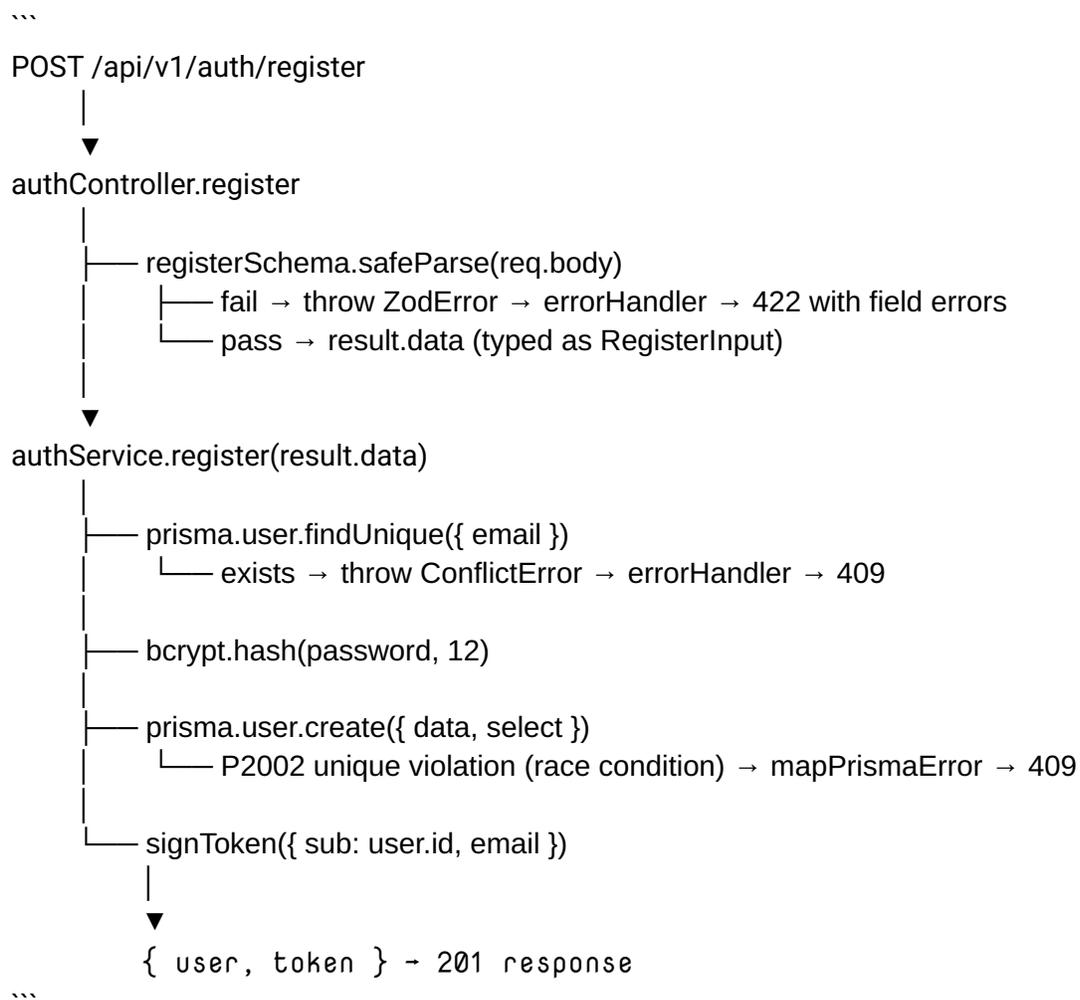
```

```

"message": "Validation failed",
"errors": {
 "email": ["Invalid email format"],
 "password": ["Password must contain at least one uppercase letter"]
}
}
...

```

## ## Data Flow



The Prisma `P2002` path handles the race condition where two concurrent registrations with the same email both pass the `findUnique` check before either `create` completes. The

``mapPrismaError`` in ``errorHandler`` catches the resulting unique constraint violation and returns a 409 rather than a 500.

**USER: Create the login endpoint. Should validate credentials and return a JWT token. Make sure it follows the same patterns as registration.**

Note: Specific factual details should be independently verified. If uncertainty exists, consult authoritative sources.

# Login Endpoint

The ``authService.login`` and ``authController.login`` methods were already stubbed in the previous step. This response completes them properly, adds a login-specific validation schema, and fills in the pieces that were left as placeholders.

--

## Files to Modify

...

```
src/
├── modules/auth/
│ ├── auth.validation.ts # Add loginSchema
│ ├── auth.service.ts # Complete login method
│ └── auth.controller.ts # Complete login handler
└──
```

No new files. No changes to routes, middleware, or error handling.

--

## ``src/modules/auth/auth.validation.ts``

Add ``loginSchema`` below the existing ``registerSchema``:

```
``typescript
import { z } from "zod";

export const registerSchema = z.object({
 email: z
 .string({ required_error: "Email is required" })
 .email("Invalid email format")
});
```

```
.toLowerCase()
.max(255, "Email must not exceed 255 characters"),
```

```
password: z
```

```
.string({ required_error: "Password is required" })
.min(8, "Password must be at least 8 characters")
.max(72, "Password must not exceed 72 characters")
.regex(/[A-Z]/, "Password must contain at least one uppercase letter")
.regex(/[0-9]/, "Password must contain at least one number"),
```

```
firstName: z
```

```
.string({ required_error: "First name is required" })
.trim()
.min(1, "First name cannot be empty")
.max(100, "First name must not exceed 100 characters"),
```

```
lastName: z
```

```
.string({ required_error: "Last name is required" })
.trim()
.min(1, "Last name cannot be empty")
.max(100, "Last name must not exceed 100 characters"),
```

```
});
```

```
export type RegisterInput = z.infertypeof registerSchema;
```

```
// — Login
```

---

```
export const loginSchema = z.object({
```

```
 email: z
```

```
 .string({ required_error: "Email is required" })
 .email("Invalid email format")
 .toLowerCase() // Normalize to match how registration stored it
 .max(255, "Email must not exceed 255 characters"),
```

```
 password: z
```

```
 .string({ required_error: "Password is required" })
 .min(1, "Password is required")
 .max(72, "Password must not exceed 72 characters"),
```

```
});
```

```
export type LoginInput = z.inferTypeof loginSchema;
...

```

**\*\*Why login password validation is looser than registration:\*\*** At login, you are verifying a credential that already exists — not enforcing a policy on a new one. Applying the uppercase/number rules at login would lock out users whose passwords were created before those rules existed, or created via a different path. The only constraints that make sense at login are: not empty, not absurdly long.

---

```
`src/modules/auth/auth.service.ts`

```

Replace the full file:

```
``typescript
import bcrypt from "bcryptjs";
import jwt from "jsonwebtoken";
import prisma from "../../config/database";
import { env } from "../../config/env";
import { ConflictError, UnauthorizedError } from "../../errors/HttpErrors";
import { JwtPayload } from "../../types";
import { RegisterInput, LoginInput } from "../auth.validation";

const SALT_ROUNDS = 12;

// Dummy hash used when a user is not found.
// Ensures bcrypt.compare always runs, preventing timing-based email enumeration.
// The value must be a valid bcrypt hash — an invalid string causes compare() to
// throw rather than return false, which would break the constant-time guarantee.
const DUMMY_HASH =
 "$2b$12$invalidhashpaddinginvalidhashpaddinginvalidhashpadXXXXX";

export const authService = {
 async register(input: RegisterInput) {
 // — 1. Check for existing user

 const existing = await prisma.user.findUnique({
 where: { email: input.email },
 select: { id: true },
 });
 }
};

```

```
if (existing) {
 throw new ConflictError(
 "An account with this email already exists",
 "EMAIL_TAKEN"
);
}
```

```
// — 2. Hash password
```

---

```
const passwordHash = await bcrypt.hash(input.password, SALT_ROUNDS);
```

```
// — 3. Create user
```

---

```
const user = await prisma.user.create({
 data: {
 email: input.email,
 passwordHash,
 firstName: input.firstName,
 lastName: input.lastName,
 },
 select: {
 id: true,
 email: true,
 firstName: true,
 lastName: true,
 createdAt: true,
 },
});
```

```
// — 4. Issue token
```

---

```
const token = authService.signToken({
 sub: user.id,
 email: user.email,
});

return { user, token };
},
```

```
async login(input: LoginInput) {
 // — 1. Look up user
```

---

```
 const user = await prisma.user.findUnique({
 where: { email: input.email },
 select: {
 id: true,
 email: true,
 passwordHash: true,
 firstName: true,
 lastName: true,
 },
 });
```

```
 // — 2. Verify password
```

---

```
 // Always run bcrypt.compare regardless of whether the user was found.
 // If we returned early on a missing user, the fast response time would
 // reveal to an attacker that the email is not registered.
 const hashToCompare = user?.passwordHash ?? DUMMY_HASH;
 const passwordValid = await bcrypt.compare(input.password, hashToCompare);
```

```
 if (!user || !passwordValid) {
 // Single message for both "no user" and "wrong password" —
 // prevents confirming which emails are registered.
 throw new UnauthorizedError(
 "Invalid email or password",
 "INVALID_CREDENTIALS"
);
 }
```

```
 // — 3. Issue token
```

---

```
 const token = authService.signToken({
 sub: user.id,
 email: user.email,
 });
```

```
 return {
 token,
```

```

 user: {
 id: user.id,
 email: user.email,
 firstName: user.firstName,
 lastName: user.lastName,
 },
 };
},

signToken(payload: JwtPayload): string {
 return jwt.sign(payload, env.JWT_SECRET, {
 expiresIn: env.JWT_EXPIRES_IN,
 });
},
};
...

```

**\*\*Why `select` includes `passwordHash` here but not in `register`:\*\*** At login, the hash is needed to call `bcrypt.compare`. It is selected explicitly and used within the service method – it is never passed to the return value. The return object is constructed manually with only the fields the caller should receive.

---

```

`src/modules/auth/auth.controller.ts`

```

Replace the full file:

```

``typescript
import { Request, Response, NextFunction } from "express";
import { registerSchema, loginSchema } from "./auth.validation";
import { authService } from "./auth.service";

export const authController = {
 async register(req: Request, res: Response, next: NextFunction): Promisevoid {
 try {
 const result = registerSchema.safeParse(req.body);

 if (!result.success) {
 throw result.error;
 }
 }
 }
};

```

```

const { user, token } = await authService.register(result.data);

res.status(201).json({
 status: "success",
 data: { user, token },
});
} catch (err) {
 next(err);
}
},

async login(req: Request, res: Response, next: NextFunction): Promisevoid {
 try {
 // — Validate

const result = loginSchema.safeParse(req.body);

if (!result.success) {
 throw result.error;
}

// — Execute

const { user, token } = await authService.login(result.data);

// — Respond

res.status(200).json({
 status: "success",
 data: { user, token },
});
} catch (err) {
 next(err);
}
},
};
...

```

The login handler is now structurally identical to register: parse → throw ZodError on failure → call service → respond. The `typeof email !== "string"` guard from the previous stub is removed — `loginSchema` handles that.

---

## ## Request / Response Reference

### `POST /api/v1/auth/login`

**\*\*Request body:\*\***

```
```json
{
  "email": "jane@example.com",
  "password": "SecurePass1"
}
...

```

****200 OK:****

```
```json
{
 "status": "success",
 "data": {
 "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
 "user": {
 "id": "clxyz123",
 "email": "jane@example.com",
 "firstName": "Jane",
 "lastName": "Doe"
 }
 }
}
...

```

**\*\*401 Unauthorized – wrong email or password:\*\***

```
```json
{
  "status": "error",
  "code": "INVALID_CREDENTIALS",
  "message": "Invalid email or password"
}
...

```

****422 Unprocessable Entity – missing or malformed fields:****

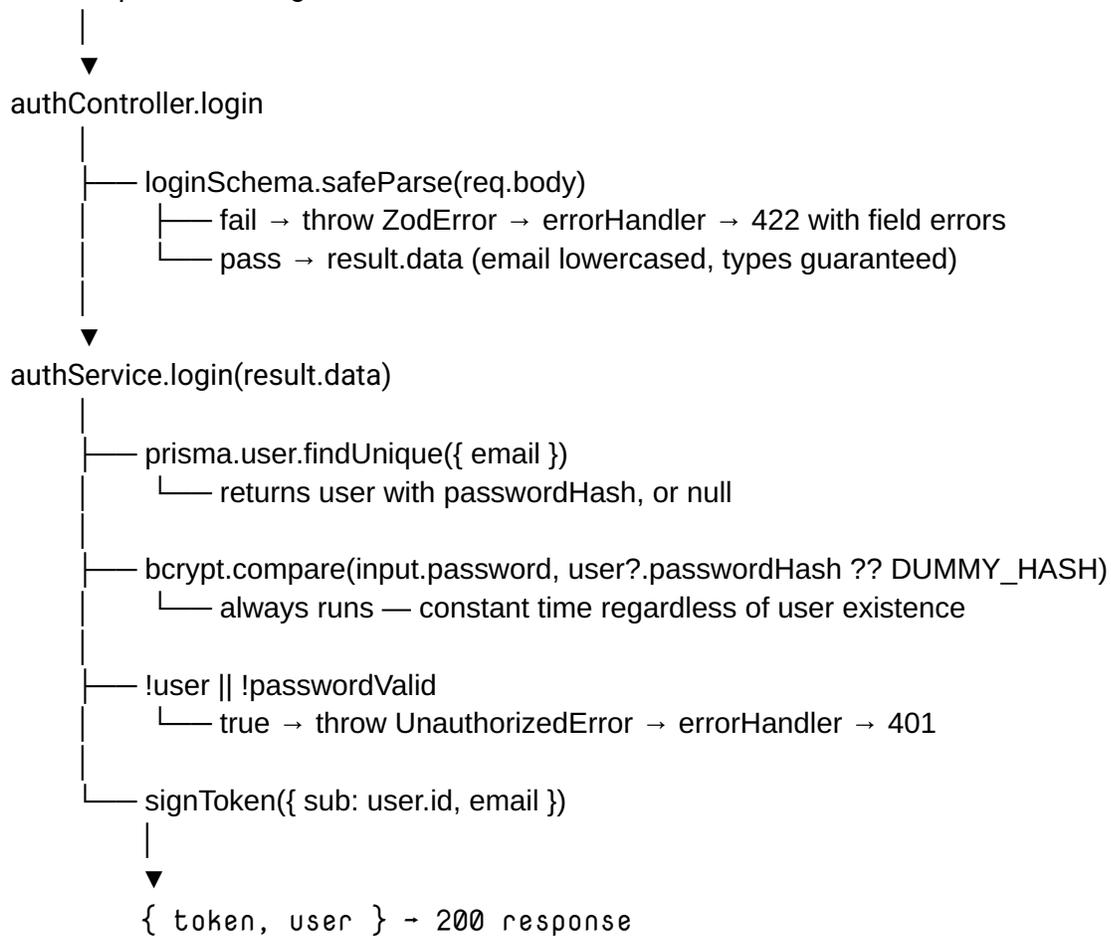
```
```json
```

```
{
 "status": "error",
 "code": "VALIDATION_ERROR",
 "message": "Validation failed",
 "errors": {
 "email": ["Invalid email format"],
 "password": ["Password is required"]
 }
}
...
--
```

## ## Data Flow

...

POST /api/v1/auth/login



...

**USER: Now build the endpoint to create an organization. Only authenticated users should be able to do this, and the creator should automatically be added as an admin of that organization.**

Note: Specific factual details should be independently verified. If uncertainty exists, consult authoritative sources.

# Create Organization Endpoint

## Files to Create/Modify

...

```
src/
├── modules/organizations/
│ ├── organizations.routes.ts # New
│ ├── organizations.controller.ts # New
│ ├── organizations.service.ts # New
│ └── organizations.validation.ts # New
└── app.ts # Mount new routes
```

...

---

## `src/modules/organizations/organizations.validation.ts`

``typescript

```
import { z } from "zod";
```

```
export const createOrganizationSchema = z.object({
 name: z
 .string({ required_error: "Organization name is required" })
 .trim()
 .min(1, "Organization name cannot be empty")
 .max(255, "Organization name must not exceed 255 characters"),
```

```
 slug: z
 .string({ required_error: "Slug is required" })
 .trim()
 .toLowerCase()
 .min(2, "Slug must be at least 2 characters")
 .max(100, "Slug must not exceed 100 characters")
```

```
.regex(
 /^[a-z0-9]+(?:-[a-z0-9]+)*$/,
 "Slug may only contain lowercase letters, numbers, and hyphens, and cannot start or end
 with a hyphen"
),
});
```

```
export type CreateOrganizationInput = z.inferTypeof createOrganizationSchema;
...
```

**\*\*Why validate slug format strictly:\*\*** The slug is used as a URL-safe identifier (e.g. `/orgs/acme-corp`). Enforcing the regex at the validation layer means the database `@@unique` constraint on `slug` only ever sees well-formed values. The regex `^[a-z0-9]+(?:-[a-z0-9]+)*$` permits `acme`, `acme-corp`, `acme-corp-2` and rejects `acme-`, `-acme`, `acme--corp`, `Acme Corp`.

---

```
`src/modules/organizations/organizations.service.ts`
```

```
``typescript
import prisma from "../../config/database";
import { ConflictError } from "../../errors/HttpErrors";
import { CreateOrganizationInput } from "../organizations.validation";
```

```
export const organizationsService = {
 async create(input: CreateOrganizationInput, creatorId: string) {
 // — 1. Check slug availability
```

---

```
 const existing = await prisma.organization.findUnique({
 where: { slug: input.slug },
 select: { id: true },
 });
```

```
 if (existing) {
 throw new ConflictError(
 `The slug "${input.slug}" is already taken`,
 "SLUG_TAKEN"
);
 }
 }
```

// — 2. Create org and membership in a single transaction —————

```
const organization = await prisma.$transaction(async (tx) => {
 const org = await tx.organization.create({
 data: {
 name: input.name,
 slug: input.slug,
 },
 select: {
 id: true,
 name: true,
 slug: true,
 createdAt: true,
 },
 });

 await tx.userOrganization.create({
 data: {
 userId: creatorId,
 organizationId: org.id,
 role: "OWNER",
 },
 });

 return org;
});

return organization;
},
};
...

```

**\*\*Why a transaction:\*\*** The organization row and the membership row must be created together. If the `userOrganization.create` fails after `organization.create` succeeds, the org would exist with no owner — an invalid state. Wrapping both in `prisma.$transaction` ensures either both writes commit or neither does.

**\*\*Why `OWNER` and not `ADMIN`:\*\*** The schema defines `OWNER` as the role with full control, including the ability to delete the organization. The creator is the owner by definition. `ADMIN` is a role granted to others by the owner afterward.

**\*\*Why check slug before the transaction:\*\*** The `@@unique` constraint on `slug` would cause a Prisma `P2002` error inside the transaction if the slug is taken, which `mapPrismaError` would convert to a 409. The explicit pre-check is done here to return a more specific error message that includes the slug value. Both paths produce a 409 – the pre-check just improves the message quality. The `P2002` fallback in `errorHandler` still handles the race condition case.

---

```
`src/modules/organizations/organizations.controller.ts`
```

```
``typescript
```

```
import { Response, NextFunction } from "express";
import { createOrganizationSchema } from "../organizations.validation";
import { organizationsService } from "../organizations.service";
import { AuthenticatedRequest } from "../../types";
```

```
export const organizationsController = {
```

```
 async create(
 req: AuthenticatedRequest,
 res: Response,
 next: NextFunction
```

```
): Promisevoid {
```

```
 try {
 // — Validate
```

---

```
 const result = createOrganizationSchema.safeParse(req.body);
```

```
 if (!result.success) {
 throw result.error;
 }
```

```
 // — Execute
```

---

```
 const organization = await organizationsService.create(
 result.data,
 req.user.sub // creatorId comes from the verified JWT – not from req.body
);
```

```
 // — Respond
```

---

```
 res.status(201).json({
 status: "success",
 data: { organization },
```

```
 });
 } catch (err) {
 next(err);
 }
},
};
...
```

**\*\*Why `req.user.sub` and not `req.body.userId`:\*\*** The creator's identity must come from the verified JWT, not from the request body. Accepting a `userId` from the body would let any authenticated user claim to be someone else when creating an organization. `req.user` is set by the `authenticate` middleware after verifying the token signature – it cannot be spoofed.

---

```
`src/modules/organizations/organizations.routes.ts`
```

```
``typescript
```

```
import { Router } from "express";
import { authenticate } from "../../middleware/authenticate";
import { organizationsController } from "../organizations.controller";
import { AuthenticatedRequest } from "../../types";
import { Request, Response, NextFunction } from "express";
```

```
const router = Router();
```

```
// Thin wrapper so Express accepts AuthenticatedRequest without a cast at every route
function asAuthenticated(
 handler: (req: AuthenticatedRequest, res: Response, next: NextFunction) = Promisevoid
) {
 return (req: Request, res: Response, next: NextFunction) =
 handler(req as AuthenticatedRequest, res, next);
}
```

```
// POST /api/v1/organizations
router.post(
 "/",
 authenticate,
 asAuthenticated(organizationsController.create)
);
```

```
export default router;
```

...

**\*\*Why `authenticate` is applied per-route and not at the router level:\*\*** Applying it at the router level would require every route in this file to be authenticated. As the organizations module grows (e.g. a public endpoint to check slug availability), per-route application keeps the intent explicit and avoids accidentally protecting routes that should be public.

---

## Updated `src/app.ts`

Add the organizations routes mount. Only the changed section is shown:

```
``typescript
import express from "express";
import { env } from "../config/env";
import { errorHandler } from "../middleware/errorHandler";
import authRoutes from "../modules/auth/auth.routes";
import organizationRoutes from "../modules/organizations/organizations.routes"; // Add

const app = express();

app.use(express.json());

app.get("/health", (req, res) = {
 res.json({ status: "ok", environment: env.NODE_ENV });
});

app.use("/api/v1/auth", authRoutes);
app.use("/api/v1/organizations", organizationRoutes); // Add

app.use(errorHandler);

app.listen(env.PORT, () = {
 console.log(`🚀 Server running on port ${env.PORT} [${env.NODE_ENV}]`);
});

export default app;
...

```

## Request / Response Reference

### `POST /api/v1/organizations`

**\*\*Headers:\*\***

...

Authorization: Bearer token

Content-Type: application/json

...

**\*\*Request body:\*\***

```json

```
{  
  "name": "Acme Corporation",  
  "slug": "acme-corp"  
}
```

...

****201 Created:****

```json

```
{
 "status": "success",
 "data": {
 "organization": {
 "id": "clxyz456",
 "name": "Acme Corporation",
 "slug": "acme-corp",
 "createdAt": "2024-01-15T10:30:00.000Z"
 }
 }
}
```

...

}

}

...

**\*\*401 Unauthorized – missing or invalid token:\*\***

```json

```
{  
  "status": "error",  
  "code": "UNAUTHORIZED",  
  "message": "Authorization header missing or malformed"  
}
```

...

...

****409 Conflict – slug already taken:****

```
```json
{
 "status": "error",
 "code": "SLUG_TAKEN",
 "message": "The slug \"acme-corp\" is already taken"
}
```
```

****422 Unprocessable Entity – validation failure:****

```
```json
{
 "status": "error",
 "code": "VALIDATION_ERROR",
 "message": "Validation failed",
 "errors": {
 "slug": ["Slug may only contain lowercase letters, numbers, and hyphens, and cannot start or end with a hyphen"]
 }
}
```
```

Data Flow

...

POST /api/v1/organizations



authenticate middleware

|— no/invalid token → 401

|— valid → req.user = { sub, email } → next()



organizationsController.create



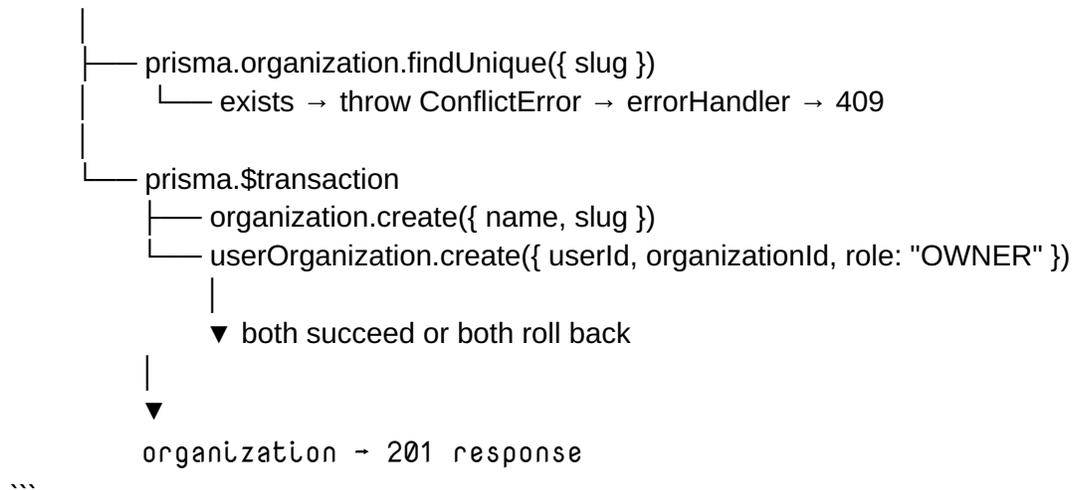
|— createOrganizationSchema.safeParse(req.body)

|— fail → throw ZodError → errorHandler → 422

|— pass → { name, slug } (slug lowercased, trimmed)



organizationsService.create(input, req.user.sub)



Optional / Future Enhancements

- **Slug auto-generation** – derive a slug from `name` if the client does not provide one (e.g. `Acme Corp` – `acme-corp`), with a uniqueness suffix if needed (`acme-corp-2`)

- **Slug availability check endpoint** – `GET

/api/v1/organizations/slug-check?slug=acme-corp` so clients can validate availability before form submission without attempting a full create

USER: Create an endpoint to add users to an organization. Only organization admins should be able to do this. Make sure we're enforcing authorization properly.

Note: Specific factual details should be independently verified. If uncertainty exists, consult authoritative sources.

Add User to Organization Endpoint

Files to Create/Modify

...

```

src/
├── modules/organizations/
│   ├── organizations.routes.ts # Add new route
│   ├── organizations.controller.ts # Add new handler
│   └── organizations.service.ts # Add new method

```

```
|  └─ organizations.validation.ts # Add new schema
...

```

```
---
```

```
## `src/modules/organizations/organizations.validation.ts`
```

Add `addMemberSchema` below the existing `createOrganizationSchema`:

```
``typescript
```

```
import { z } from "zod";
```

```
import { OrgRole } from "@prisma/client";
```

```
export const createOrganizationSchema = z.object({
```

```
  name: z
```

```
    .string({ required_error: "Organization name is required" })
```

```
    .trim()
```

```
    .min(1, "Organization name cannot be empty")
```

```
    .max(255, "Organization name must not exceed 255 characters"),
```

```
  slug: z
```

```
    .string({ required_error: "Slug is required" })
```

```
    .trim()
```

```
    .toLowerCase()
```

```
    .min(2, "Slug must be at least 2 characters")
```

```
    .max(100, "Slug must not exceed 100 characters")
```

```
    .regex(
```

```
      /^[a-z0-9]+(?:-[a-z0-9]+)*$/,
```

```
      "Slug may only contain lowercase letters, numbers, and hyphens, and cannot start or end  
with a hyphen"
```

```
    ),
```

```
  });
```

```
export type CreateOrganizationInput = z.infer
```

```
// — Add Member
```

```
export const addMemberSchema = z.object({
```

```
  email: z
```

```
    .string({ required_error: "Email is required" })
```

```

    .email("Invalid email format")
    .toLowerCase()
    .max(255, "Email must not exceed 255 characters"),

role: z
  .enum(["ADMIN", "MEMBER"] as const, {
    errorMap: () = ({ message: "Role must be ADMIN or MEMBER" }),
  })
  .default("MEMBER"),
});

export type AddMemberInput = z.infer;
...

```

****Why `OWNER` is excluded from the assignable roles:**** `OWNER` is granted exactly once – at org creation. It represents the founding principal and carries delete/billing authority. Allowing admins to assign `OWNER` to arbitrary users would be a privilege escalation path. Ownership transfer, if needed, is a separate operation with its own semantics and is out of scope here.

```
## `src/modules/organizations/organizations.service.ts`
```

Add `addMember` to the existing service object:

```

``typescript
import prisma from "../../config/database";
import { ConflictError, NotFoundError } from "../../errors/HttpErrors";
import { CreateOrganizationInput, AddMemberInput } from "../organizations.validation";
import { OrgRole } from "@prisma/client";

export const organizationsService = {
  async create(input: CreateOrganizationInput, creatorId: string) {
    const existing = await prisma.organization.findUnique({
      where: { slug: input.slug },
      select: { id: true },
    });

    if (existing) {
      throw new ConflictError(
        `The slug "${input.slug}" is already taken`,
        "SLUG_TAKEN"
      );
    }
  }
};

```

```
);  
}
```

```
const organization = await prisma.$transaction(async (tx) => {  
  const org = await tx.organization.create({  
    data: { name: input.name, slug: input.slug },  
    select: { id: true, name: true, slug: true, createdAt: true },  
  });
```

```
  await tx.userOrganization.create({  
    data: { userId: creatorId, organizationId: org.id, role: "OWNER" },  
  });
```

```
  return org;  
});
```

```
return organization;  
},
```

```
async addMember(  
  organizationId: string,  
  input: AddMemberInput,  
  actorRole: OrgRole  
) {
```

```
  // — 1. Enforce role assignment rules —————
```

```
  // ADMIN can only assign MEMBER. OWNER can assign ADMIN or MEMBER.  
  // This is checked here in the service rather than only in middleware  
  // because the rule depends on the combination of actor role + target role.  
  if (actorRole === "ADMIN" && input.role === "ADMIN") {  
    throw new ForbiddenError(  
      "Admins cannot assign the ADMIN role. Only owners can.",  
      "INSUFFICIENT_ROLE"  
    );  
  }
```

```
  // — 2. Resolve the target user by email —————
```

```
  const targetUser = await prisma.user.findUnique({  
    where: { email: input.email },  
    select: { id: true, email: true, firstName: true, lastName: true },  
  });
```

```
if (!targetUser) {  
  // Vague message – do not confirm whether an email is registered  
  throw new NotFoundError("User", "USER_NOT_FOUND");  
}
```

```
// — 3. Check if already a member
```

```
const existingMembership = await prisma.userOrganization.findUnique({  
  where: {  
    userId_organizationId: {  
      userId: targetUser.id,  
      organizationId,  
    },  
  },  
  select: { role: true },  
});
```

```
if (existingMembership) {  
  throw new ConflictError(  
    "This user is already a member of the organization",  
    "ALREADY_A_MEMBER"  
  );  
}
```

```
// — 4. Create membership
```

```
const membership = await prisma.userOrganization.create({  
  data: {  
    userId: targetUser.id,  
    organizationId,  
    role: input.role,  
  },  
  select: {  
    role: true,  
    joinedAt: true,  
  },  
});
```

```
return {
```

```
    user: targetUser,
    role: membership.role,
    joinedAt: membership.joinedAt,
  };
},
};
...
```

****Why look up the target user by email rather than accepting a `userId`:**** Admins know their colleagues' email addresses – they do not know internal database IDs. Accepting an email is the correct UX. The service resolves it to an ID internally.

****Why the vague "User not found" message when the email is unregistered:**** Returning "No account exists for this email" would let an org admin enumerate which emails have accounts on the platform. The vague message reveals nothing about whether the email is registered.

****Why check `ALREADY_A_MEMBER` before creating:**** The `@@unique([userId, organizationId])` constraint would catch this as a `P2002` error, which `mapPrismaError` converts to a generic 409. The explicit pre-check returns a more specific `ALREADY_A_MEMBER` code that clients can act on meaningfully (e.g. offer a role-update flow instead).

****Why the ADMIN-cannot-assign-ADMIN rule lives in the service:**** The `requireRole("ADMIN")` middleware only checks that the actor has *at least* ADMIN access. It does not know what role is being assigned. The combination of actor role + requested role is business logic – it belongs in the service, not in generic middleware.

```
## `src/modules/organizations/organizations.controller.ts`
```

Add `addMember` to the existing controller object:

```
``typescript
import { Response, NextFunction } from "express";
import { createOrganizationSchema, addMemberSchema } from "./organizations.validation";
import { organizationsService } from "./organizations.service";
import { AuthenticatedRequest, OrgScopedRequest } from "../../types";

export const organizationsController = {
  async create(
    req: AuthenticatedRequest,
```

```
res: Response,  
next: NextFunction  
) : Promisevoid {  
  try {  
    const result = createOrganizationSchema.safeParse(req.body);  
    if (!result.success) throw result.error;  
  
    const organization = await organizationsService.create(  
      result.data,  
      req.user.sub  
    );  
  
    res.status(201).json({ status: "success", data: { organization } });  
  } catch (err) {  
    next(err);  
  }  
},
```

```
async addMember(  
  req: OrgScopedRequest,  
  res: Response,  
  next: NextFunction  
) : Promisevoid {  
  try {  
    // — Validate
```

```
    const result = addMemberSchema.safeParse(req.body);  
    if (!result.success) throw result.error;
```

```
    // — Execute
```

```
    // organizationId and actorRole come from req.membership,  
    // set by requireRole middleware – not from req.body or req.params  
    const membership = await organizationsService.addMember(  
      req.membership.organizationId,  
      result.data,  
      req.membership.role  
    );
```

```
    // — Respond
```

```
    res.status(201).json({ status: "success", data: { membership } });  
  } catch (err) {
```

```
    next(err);
  }
},
};
...

```

****Why `req.membership.organizationId` instead of `req.params.organizationId`:**** Both hold the same value at runtime, but `req.membership.organizationId` is the value that was verified by the database lookup inside `requireRole`. Using it makes the data provenance explicit – this ID was confirmed to exist and the actor was confirmed to be a member of it.

--

```
## `src/modules/organizations/organizations.routes.ts`
```

Replace the full file:

```
``typescript
import { Router } from "express";
import { Request, Response, NextFunction } from "express";
import { authenticate } from "../../middleware/authenticate";
import { requireRole } from "../../middleware/requireRole";
import { organizationsController } from "../organizations.controller";
import { AuthenticatedRequest, OrgScopedRequest } from "../../types";

const router = Router();

function asAuthenticated(
  handler: (req: AuthenticatedRequest, res: Response, next: NextFunction) = Promisevoid
){
  return (req: Request, res: Response, next: NextFunction) =
    handler(req as AuthenticatedRequest, res, next);
}

function asOrgScoped(
  handler: (req: OrgScopedRequest, res: Response, next: NextFunction) = Promisevoid
){
  return (req: Request, res: Response, next: NextFunction) =
    handler(req as OrgScopedRequest, res, next);
}

// POST /api/v1/organizations

```

```

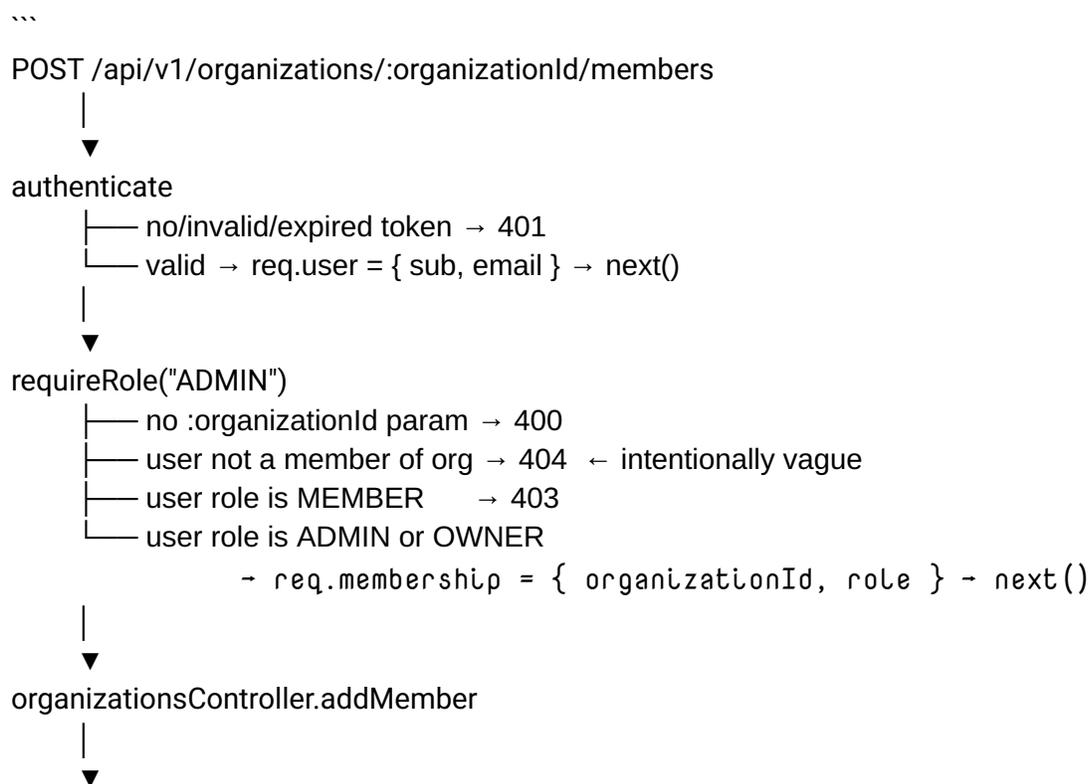
// Any authenticated user can create an organization
router.post(
  "/",
  authenticate,
  asAuthenticated(organizationsController.create)
);

// POST /api/v1/organizations/:organizationId/members
// Only ADMIN or OWNER of the organization can add members
router.post(
  ":",organizationId/members",
  authenticate,
  requireRole("ADMIN"),
  asOrgScoped(organizationsController.addMember)
);

export default router;
...

```

Authorization Layer Diagram



organizationsService.addMember(organizationId, input, actorRole)

```
|
|— actorRole is ADMIN and input.role is ADMIN → 403
|— target email not found → 404
|— target already a member → 409
|— create UserOrganization row → 201
...

```

The authorization check spans two layers intentionally:

| Layer | What it enforces |

|---|---|

| `requireRole("ADMIN")` | Actor is a member of this org with ADMIN or OWNER role |

| `organizationsService.addMember` | Actor's specific role permits the requested role assignment |

Neither layer alone is sufficient. The middleware does not know what role is being assigned. The service does not re-verify org membership – it trusts that middleware already did.

Request / Response Reference

`POST /api/v1/organizations/:organizationId/members`

****Headers:****

...

Authorization: Bearer token

Content-Type: application/json

...

****Request body:****

```json

{

"email": "bob@example.com",

"role": "MEMBER"

}

...

**\*\*201 Created:\*\***

```json

{

"status": "success",

```
"data": {
  "membership": {
    "user": {
      "id": "clxyz789",
      "email": "bob@example.com",
      "firstName": "Bob",
      "lastName": "Smith"
    },
    "role": "MEMBER",
    "joinedAt": "2024-01-15T11:00:00.000Z"
  }
}
...
```

****403 Forbidden – MEMBER attempting to add someone:****

```
``json
{
  "status": "error",
  "code": "INSUFFICIENT_ROLE",
  "message": "This action requires ADMIN role or higher"
}
...
```

****403 Forbidden – ADMIN attempting to assign ADMIN role:****

```
``json
{
  "status": "error",
  "code": "INSUFFICIENT_ROLE",
  "message": "Admins cannot assign the ADMIN role. Only owners can."
}
...
```

****404 Not Found – target email not registered:****

```
``json
{
  "status": "error",
  "code": "USER_NOT_FOUND",
  "message": "User not found"
}
...
```

****409 Conflict – user already a member:****

```
```json
{
 "status": "error",
 "code": "ALREADY_A_MEMBER",
 "message": "This user is already a member of the organization"
}
```
```

Optional / Future Enhancements

- ****Invitation flow**** – instead of adding users directly by email, send an invitation link that the target user must accept, giving them agency over which orgs they join
- ****Role update endpoint**** – `PATCH /organizations/:organizationId/members/:userId` to change an existing member's role, with the same ADMIN-cannot-promote-to-ADMIN rule
- ****Remove member endpoint**** – `DELETE /organizations/:organizationId/members/:userId`, with a guard preventing the last OWNER from being removed

USER:Build the CRUD endpoints for tasks. Tasks should:

- **Belong to an organization**
- **Have a title, description, status (todo/in_progress/done), and assignee**
- **Only be accessible by members of that organization**

Show me all four endpoints (create, read, update, delete).

Note: Specific factual details should be independently verified. If uncertainty exists, consult authoritative sources.

Task CRUD Endpoints

Files to Create

```
...
src/
├── modules/tasks/
│   ├── tasks.routes.ts
│   ├── tasks.controller.ts
│   ├── tasks.service.ts
│   └── tasks.validation.ts
└── app.ts # Mount new routes
...
```

```
## `src/modules/tasks/tasks.validation.ts`
```

```
``typescript
```

```
import { z } from "zod";
```

```
import { TaskStatus, TaskPriority } from "@prisma/client";
```

```
export const createTaskSchema = z.object({
```

```
  title: z
```

```
    .string({ required_error: "Title is required" })
```

```
    .trim()
```

```
    .min(1, "Title cannot be empty")
```

```
    .max(500, "Title must not exceed 500 characters"),
```

```
  description: z
```

```
    .string()
```

```
    .trim()
```

```
    .max(10000, "Description must not exceed 10000 characters")
```

```
    .optional(),
```

```
  status: z
```

```
    .nativeEnum(TaskStatus, {
```

```
      errorMap: () = ({ message: "Status must be TODO, IN_PROGRESS, or DONE" }),
```

```
    })
```

```
    .default("TODO"),
```

```
  priority: z
```

```
    .nativeEnum(TaskPriority, {
```

```
      errorMap: () = ({ message: "Priority must be LOW, MEDIUM, or HIGH" }),
```

```
    })
```

```
    .default("MEDIUM"),
```

```
  assigneeId: z
```

```
    .string()
```

```
    .cuid("Assignee ID must be a valid ID")
```

```
    .optional()
```

```
    .nullable(),
```

```
  dueDate: z
```

```
    .string()
```

```

.datetime({ message: "Due date must be a valid ISO 8601 datetime" })
.optional()
.nullable()
.transform((val) = (val ? new Date(val) : null)),
});

// Update allows all fields to be optional – only provided fields are changed
export const updateTaskSchema = createTaskSchema
.omit({ status: true, priority: true })
.extend({
  status: z
    .nativeEnum(TaskStatus, {
      errorMap: () = ({ message: "Status must be TODO, IN_PROGRESS, or DONE" }),
    })
    .optional(),

  priority: z
    .nativeEnum(TaskPriority, {
      errorMap: () = ({ message: "Priority must be LOW, MEDIUM, or HIGH" }),
    })
    .optional(),
})
.partial();

export const listTasksQuerySchema = z.object({
  status: z
    .nativeEnum(TaskStatus, {
      errorMap: () = ({ message: "Status must be TODO, IN_PROGRESS, or DONE" }),
    })
    .optional(),

  assigneeId: z.string().cuid().optional(),

  page: z
    .string()
    .optional()
    .transform((val) = (val ? parseInt(val, 10) : 1))
    .pipe(z.number().int().min(1, "Page must be at least 1")),

  limit: z
    .string()
    .optional()

```

```
.transform((val) = (val ? parseInt(val, 10) : 20))
.pipe(z.number().int().min(1).max(100, "Limit must not exceed 100")),
});
```

```
export type CreateTaskInput = z.infer
```

****Why `z.nativeEnum` instead of `z.enum`:**** `z.nativeEnum` accepts the Prisma-generated enum directly. `z.enum` requires a string tuple literal. Using `nativeEnum` means the validation and the database type are always in sync — adding a new status to the Prisma schema automatically makes it valid here without touching the validation file.

****Why `updateTaskSchema` uses `.partial()`:**** A PATCH endpoint should only update the fields the client sends. `.partial()` makes every field optional so the service can apply only the provided changes. A PUT (full replacement) would use the full `createTaskSchema` instead.

```
## `src/modules/tasks/tasks.service.ts`
```

```
````typescript
import prisma from "../../config/database";
import { NotFoundError, ForbiddenError } from "../../errors/HttpErrors";
import { CreateTaskInput, UpdateTaskInput, ListTasksQuery } from "./tasks.validation";

// Fields returned on every task response — passwordHash equivalent excluded by design
const TASK_SELECT = {
 id: true,
 title: true,
 description: true,
 status: true,
 priority: true,
 dueDate: true,
 createdAt: true,
 updatedAt: true,
 organizationId: true,
 creator: {
 select: { id: true, firstName: true, lastName: true, email: true },
 },
 assignee: {
```

```

 select: { id: true, firstName: true, lastName: true, email: true },
 },
} as const;

export const tasksService = {
 async create(
 organizationId: string,
 creatorId: string,
 input: CreateTaskInput
) {
 // If an assigneeId is provided, verify they are a member of this org
 if (input.assigneeId) {
 await assertOrgMember(input.assigneeId, organizationId);
 }

 const task = await prisma.task.create({
 data: {
 title: input.title,
 description: input.description,
 status: input.status,
 priority: input.priority,
 dueDate: input.dueDate,
 organizationId,
 creatorId,
 assigneeId: input.assigneeId ?? null,
 },
 select: TASK_SELECT,
 });

 return task;
 },

 async list(organizationId: string, query: ListTasksQuery) {
 const { status, assigneeId, page, limit } = query;
 const skip = (page - 1) * limit;

 // Build where clause – organizationId is always present
 const where = {
 organizationId,
 ...(status !== undefined && { status }),
 ...(assigneeId !== undefined && { assigneeId }),
 };
 },
};

```

```

// Run count and data fetch in parallel
const [total, tasks] = await Promise.all([
 prisma.task.count({ where }),
 prisma.task.findMany({
 where,
 select: TASK_SELECT,
 orderBy: { createdAt: "desc" },
 skip,
 take: limit,
 }),
]);

return {
 tasks,
 pagination: {
 total,
 page,
 limit,
 totalPages: Math.ceil(total / limit),
 },
};
},

async getById(taskId: string, organizationId: string) {
 const task = await prisma.task.findUnique({
 where: { id: taskId },
 select: { ...TASK_SELECT, organizationId: true },
 });

 // Check existence and org ownership in one place
 if (!task || task.organizationId !== organizationId) {
 throw new NotFoundError("Task", "TASK_NOT_FOUND");
 }

 return task;
},

async update(
 taskId: string,
 organizationId: string,
 input: UpdateTaskInput

```

```

) {
 // Verify task exists and belongs to this org before updating
 await tasksService.getByld(taskId, organizationId);

 // If assigneeId is being changed, verify the new assignee is an org member
 if (input.assigneeId !== undefined && input.assigneeId !== null) {
 await assertOrgMember(input.assigneeId, organizationId);
 }

 const task = await prisma.task.update({
 where: { id: taskId },
 data: {
 ...(input.title !== undefined && { title: input.title }),
 ...(input.description !== undefined && { description: input.description }),
 ...(input.status !== undefined && { status: input.status }),
 ...(input.priority !== undefined && { priority: input.priority }),
 ...(input.dueDate !== undefined && { dueDate: input.dueDate }),
 // assigneeId: undefined means "not provided" – don't touch it
 // assigneeId: null means "explicitly unassign" – set to null
 ...(input.assigneeId !== undefined && { assigneeId: input.assigneeId }),
 },
 select: TASK_SELECT,
 });

 return task;
},

async delete(taskId: string, organizationId: string) {
 // Verify task exists and belongs to this org before deleting
 await tasksService.getByld(taskId, organizationId);

 await prisma.task.delete({ where: { id: taskId } });
},
};

```

// — Private helpers

---

```

async function assertOrgMember(
 userId: string,
 organizationId: string
): Promisevoid {

```

```

const membership = await prisma.userOrganization.findUnique({
 where: {
 userId_organizationId: { userId, organizationId },
 },
 select: { userId: true },
});

if (!membership) {
 throw new ForbiddenError(
 "Assignee must be a member of this organization",
 "ASSIGNEE_NOT_MEMBER"
);
}
}
}
...

```

**\*\*Why `assertOrgMember` before creating/updating with an assignee:\*\*** Without this check, a task could be assigned to a user from a different organization. The `assigneeId` FK only enforces that the user exists – it does not enforce org membership. This is a multi-tenancy boundary that must be enforced in application code.

**\*\*Why `!task || task.organizationId !== organizationId` in `getById`:\*\*** A task with the right `taskId` but a different `organizationId` should return 404, not 403. Returning 403 would confirm to a member of org A that a task with that ID exists in some other org. The combined check leaks nothing.

**\*\*Why spread `input` conditionally in `update`:\*\*** Prisma's `update` with `undefined` values leaves those columns unchanged. However, being explicit about which fields are included makes the intent clear and avoids accidentally passing `undefined` for `assigneeId` when the client meant to unassign (send `null`). The `undefined` vs `null` distinction is meaningful here: `undefined` = "client did not mention this field", `null` = "client explicitly wants to clear it".

**\*\*Why `Promise.all` in `list`:\*\*** The `count` and `findMany` queries are independent. Running them in parallel halves the round-trip time for paginated list responses.

---

```
`src/modules/tasks/tasks.controller.ts`
```

```

``typescript
import { Response, NextFunction } from "express";
import {

```

```

 createTaskSchema,
 updateTaskSchema,
 listTasksQuerySchema,
 } from "./tasks.validation";
import { tasksService } from "./tasks.service";
import { OrgScopedRequest } from "../../types";

export const tasksController = {
 async create(
 req: OrgScopedRequest,
 res: Response,
 next: NextFunction
): Promisevoid {
 try {
 const result = createTaskSchema.safeParse(req.body);
 if (!result.success) throw result.error;

 const task = await tasksService.create(
 req.membership.organizationId,
 req.user.sub,
 result.data
);

 res.status(201).json({ status: "success", data: { task } });
 } catch (err) {
 next(err);
 }
 },

 async list(
 req: OrgScopedRequest,
 res: Response,
 next: NextFunction
): Promisevoid {
 try {
 const result = listTasksQuerySchema.safeParse(req.query);
 if (!result.success) throw result.error;

 const { tasks, pagination } = await tasksService.list(
 req.membership.organizationId,
 result.data
);
 }
 }
};

```

```
 res.status(200).json({ status: "success", data: { tasks, pagination } });
 } catch (err) {
 next(err);
 }
},
```

```
async getByld(
 req: OrgScopedRequest,
 res: Response,
 next: NextFunction
): Promisevoid {
 try {
 const task = await tasksService.getByld(
 req.params.taskId,
 req.membership.organizationId
);

 res.status(200).json({ status: "success", data: { task } });
 } catch (err) {
 next(err);
 }
},
```

```
async update(
 req: OrgScopedRequest,
 res: Response,
 next: NextFunction
): Promisevoid {
 try {
 const result = updateTaskSchema.safeParse(req.body);
 if (!result.success) throw result.error;

 const task = await tasksService.update(
 req.params.taskId,
 req.membership.organizationId,
 result.data
);

 res.status(200).json({ status: "success", data: { task } });
 } catch (err) {
 next(err);
 }
},
```

```

 }
 },

 async delete(
 req: OrgScopedRequest,
 res: Response,
 next: NextFunction
): Promisevoid {
 try {
 await tasksService.delete(
 req.params.taskId,
 req.membership.organizationId
);

 res.status(204).send();
 } catch (err) {
 next(err);
 }
 },
};
...

```

**\*\*Why 204 with no body on delete:\*\*** HTTP 204 No Content is the correct status for a successful delete. Sending a body with 204 is technically invalid per the HTTP spec – the response must have no content.

---

```
`src/modules/tasks/tasks.routes.ts`
```

```
``typescript
```

```

import { Router } from "express";
import { Request, Response, NextFunction } from "express";
import { authenticate } from "../../middleware/authenticate";
import { requireRole } from "../../middleware/requireRole";
import { tasksController } from "../tasks.controller";
import { OrgScopedRequest } from "../../types";

```

```

const router = Router({ mergeParams: true }); // Required to access :organizationId from parent
router

```

```

function asOrgScoped(

```

```
 handler: (req: OrgScopedRequest, res: Response, next: NextFunction) = Promisevoid
) {
 return (req: Request, res: Response, next: NextFunction) =
 handler(req as OrgScopedRequest, res, next);
}
```

```
// All task routes require authentication and org membership
router.use(authenticate, requireRole("MEMBER"));
```

```
// GET /api/v1/organizations/:organizationId/tasks
router.get("/", asOrgScoped(tasksController.list));
```

```
// POST /api/v1/organizations/:organizationId/tasks
router.post("/", asOrgScoped(tasksController.create));
```

```
// GET /api/v1/organizations/:organizationId/tasks/:taskId
router.get("/:taskId", asOrgScoped(tasksController.getById));
```

```
// PATCH /api/v1/organizations/:organizationId/tasks/:taskId
router.patch("/:taskId", asOrgScoped(tasksController.update));
```

```
// DELETE /api/v1/organizations/:organizationId/tasks/:taskId
// Requires ADMIN or OWNER – members cannot delete tasks
router.delete("/:taskId", requireRole("ADMIN"), asOrgScoped(tasksController.delete));
```

```
export default router;
```

```
...
```

**\*\*Why `router.use(authenticate, requireRole("MEMBER"))` at the router level:\*\*** Every task endpoint requires at minimum authentication and org membership. Applying both at the router level avoids repeating them on each route. The delete route then adds a second `requireRole("ADMIN")` call – Express runs middleware in order, so the MEMBER check runs first, then the ADMIN check runs second for that route only.

**\*\*Why `mergeParams: true`:\*\*** Task routes are mounted under `/organizations/:organizationId/tasks`. Without `mergeParams: true`, `req.params.organizationId` would be undefined inside the task router because child routers do not inherit parent params by default in Express.

```

```

```
Updated `src/app.ts`
```

```

``typescript
import express from "express";
import { env } from "./config/env";
import { errorHandler } from "./middleware/errorHandler";
import authRoutes from "./modules/auth/auth.routes";
import organizationRoutes from "./modules/organizations/organizations.routes";
import taskRoutes from "./modules/tasks/tasks.routes"; // Add

const app = express();

app.use(express.json());

app.get("/health", (req, res) = {
 res.json({ status: "ok", environment: env.NODE_ENV });
});

app.use("/api/v1/auth", authRoutes);
app.use("/api/v1/organizations", organizationRoutes);

// Nested under organizations — :organizationId is available via mergeParams
app.use(
 "/api/v1/organizations/:organizationId/tasks",
 taskRoutes
); // Add

app.use(errorHandler);

app.listen(env.PORT, () = {
 console.log(`🚀 Server running on port ${env.PORT} [${env.NODE_ENV}]`);
});

export default app;
``

Request / Response Reference

`POST /api/v1/organizations/:organizationId/tasks`

Request:

```

```
```json
{
  "title": "Write API documentation",
  "description": "Cover all endpoints with examples",
  "status": "TODO",
  "priority": "HIGH",
  "assigneeId": "clxyz789",
  "dueDate": "2024-03-01T00:00:00.000Z"
}
```
```

**\*\*201 Created:\*\***

```
```json
{
  "status": "success",
  "data": {
    "task": {
      "id": "cltask123",
      "title": "Write API documentation",
      "description": "Cover all endpoints with examples",
      "status": "TODO",
      "priority": "HIGH",
      "dueDate": "2024-03-01T00:00:00.000Z",
      "organizationId": "clorg456",
      "createdAt": "2024-01-15T12:00:00.000Z",
      "updatedAt": "2024-01-15T12:00:00.000Z",
      "creator": { "id": "clxyz123", "firstName": "Jane", "lastName": "Doe", "email":
"jane@example.com" },
      "assignee": { "id": "clxyz789", "firstName": "Bob", "lastName": "Smith", "email":
"bob@example.com" }
    }
  }
}
```
```

---

### `GET /api/v1/organizations/:organizationId/tasks`

**\*\*Query params:\*\*** `?status=TODO&page=1&limit=20`

**\*\*200 OK:\*\***

```
```json
{
  "status": "success",
  "data": {
    "tasks": [/* array of task objects */],
    "pagination": {
      "total": 42,
      "page": 1,
      "limit": 20,
      "totalPages": 3
    }
  }
}
...
--
```

`GET /api/v1/organizations/:organizationId/tasks/:taskId`

****200 OK:****

```
```json
{
 "status": "success",
 "data": { "task": {/* full task object */} }
}
...
--
```

**\*\*404 Not Found – task does not exist or belongs to another org:\*\***

```
```json
{
  "status": "error",
  "code": "TASK_NOT_FOUND",
  "message": "Task not found"
}
...
--
```

`PATCH /api/v1/organizations/:organizationId/tasks/:taskId`

****Request**** – only include fields you want to change:

```
```json
```

```
{
 "status": "IN_PROGRESS",
 "assigneeld": null
}
...
```

**\*\*200 OK:\*\***

```
```json
{
  "status": "success",
  "data": {
    "task": {
      "id": "cltask123",
      "title": "Write API documentation",
      "description": "Cover all endpoints with examples",
      "status": "IN_PROGRESS",
      "priority": "HIGH",
      "dueDate": "2024-03-01T00:00:00.000Z",
      "organizationId": "clorg456",
      "createdAt": "2024-01-15T12:00:00.000Z",
      "updatedAt": "2024-01-15T12:30:00.000Z",
      "creator": { "id": "clxyz123", "firstName": "Jane", "lastName": "Doe", "email":
"jane@example.com" },
      "assignee": null
    }
  }
}
...
```

****400 Bad Request – validation failure:****

```
```json
{
 "status": "error",
 "code": "VALIDATION_ERROR",
 "message": "Validation failed",
 "errors": [
 { "field": "status", "message": "Status must be TODO, IN_PROGRESS, or DONE" }
]
}
...
```

--

### `DELETE /api/v1/organizations/:organizationId/tasks/:taskId`

**\*\*204 No Content\*\*** – empty body on success.

**\*\*403 Forbidden** – caller is MEMBER, not ADMIN or OWNER:

```
``json
{
 "status": "error",
 "code": "FORBIDDEN",
 "message": "Insufficient role"
}
...

```

**\*\*404 Not Found** – task does not exist or belongs to another org:

```
``json
{
 "status": "error",
 "code": "TASK_NOT_FOUND",
 "message": "Task not found"
}
...

```

---

## ## Edge Cases and Behavior Summary

| Scenario | Behavior |

|---|---|

| `assigneId` not in request body | Field is untouched on update |

| `assigneId: null` in request body | Task is explicitly unassigned |

| `assigneId` belongs to a different org | `403 ASSIGNEE\_NOT\_MEMBER` |

| `taskId` exists but in a different org | `404 TASK\_NOT\_FOUND` |

| `page` or `limit` omitted from query | Defaults to `page=1`, `limit=20` |

| `limit` exceeds 100 | `400 VALIDATION\_ERROR` |

| MEMBER attempts delete | `403 FORBIDDEN` |

| ADMIN or OWNER deletes | `204 No Content` |

| `dueDate: null` in request body | Due date is explicitly cleared |

| `dueDate` omitted from request body | Due date is untouched on update |

---

```
Type: `OrgScopedRequest`
```

If this type does not already exist in your project, add it to `src/types/index.ts`:

```
``typescript
import { Request } from "express";

export interface OrgScopedRequest extends Request {
 user: {
 sub: string; // userId from JWT
 email: string;
 };
 membership: {
 organizationId: string;
 role: "OWNER" | "ADMIN" | "MEMBER";
 };
}
...

```

This type is populated by the `authenticate` middleware (which attaches `req.user`) and the `requireRole` middleware (which attaches `req.membership` after verifying the user's role in the organization). Both must run before any controller handler that uses `OrgScopedRequest`.

**USER: I need pagination for the tasks list endpoint. Add pagination with limit and offset parameters, and return metadata about total count and pages.**

Note: Specific factual details should be independently verified. If uncertainty exists, consult authoritative sources.

The `listTasksQuerySchema` and `tasksService.list` already implement pagination using `page` and `limit` with full metadata. This response migrates from `page`-based to `limit`/`offset` parameters as requested, and fills in any gaps.

--

```
Files to Modify
```

```
...
```

```
src/
├── modules/tasks/
│ └── tasks.validation.ts # Replace page with offset
```

```
└── tasks.service.ts # Update list method
...

```

No other files change.

---

```
`src/modules/tasks/tasks.validation.ts`
```

Replace `listTasksQuerySchema` only. All other schemas are unchanged:

```
``typescript
export const listTasksQuerySchema = z.object({
 status: z
 .nativeEnum(TaskStatus, {
 errorMap: () = ({ message: "Status must be TODO, IN_PROGRESS, or DONE" }),
 })
 .optional(),

 assigneeId: z.string().cuid("Assignee ID must be a valid ID").optional(),

 limit: z
 .string()
 .optional()
 .transform((val) = (val !== undefined ? parseInt(val, 10) : 20))
 .pipe(
 z.number().int("Limit must be an integer").min(1, "Limit must be at least 1").max(100, "Limit
must not exceed 100")
),

 offset: z
 .string()
 .optional()
 .transform((val) = (val !== undefined ? parseInt(val, 10) : 0))
 .pipe(
 z.number().int("Offset must be an integer").min(0, "Offset must be zero or greater")
),
});

export type ListTasksQuery = z.inferTypeof listTasksQuerySchema;
...

```

**\*\*Why `page` is removed entirely:\*\*** The previous schema had `page` and derived `skip` inside the service. The request asked for `limit` and `offset` directly. Keeping both would create two ways to express the same thing, which causes ambiguity when both are provided. `offset` is the explicit, lower-level primitive – clients that want page-style navigation can compute `offset = (page - 1) \* limit` themselves.

**\*\*Why `parseInt(val, 10)` with an explicit radix:\*\*** Without the radix argument, strings beginning with `0` could be parsed as octal in older environments. Always passing `10` is a defensive habit.

---

```
`src/modules/tasks/tasks.service.ts`
```

Replace the `list` method only. All other methods are unchanged:

```
``typescript
async list(organizationId: string, query: ListTasksQuery) {
 const { status, assigneeld, limit, offset } = query;

 const where = {
 organizationId,
 ...(status !== undefined && { status }),
 ...(assigneeld !== undefined && { assigneeld }),
 };

 const [total, tasks] = await Promise.all([
 prisma.task.count({ where }),
 prisma.task.findMany({
 where,
 select: TASK_SELECT,
 orderBy: { createdAt: "desc" },
 skip: offset,
 take: limit,
 }),
]);

 const hasNextPage = offset + tasks.length > total;
 const hasPreviousPage = offset > 0;

 return {
 tasks,
```

```

 pagination: {
 total,
 limit,
 offset,
 hasNextPage,
 hasPreviousPage,
 },
 };
},
...

```

**\*\*Why `offset + tasks.length` instead of `offset + limit` total:\*\*** If the final page has fewer items than `limit`, `offset + limit` would exceed `total` but `hasNextPage` would still be `false`. Using `tasks.length` is accurate regardless of whether the current page is full.

**\*\*Why no `totalPages` in the response:\*\*** `totalPages` is a derived value that only makes sense in page-based navigation. With offset-based pagination, clients navigate by incrementing `offset` by `limit`. Including `totalPages` would imply a page model that no longer exists. Clients that need it can compute `Math.ceil(total / limit)` from the returned values.

---

## Response Shape

### `GET /api/v1/organizations/:organizationId/tasks`

**\*\*Query params:\*\*** `?status=TODO&limit=10&offset=20`

**\*\*200 OK:\*\***

```

````json
{
  "status": "success",
  "data": {
    "tasks": [ /* array of task objects */ ],
    "pagination": {
      "total": 42,
      "limit": 10,
      "offset": 20,
      "hasNextPage": true,
      "hasPreviousPage": true
    }
  }
}

```

```
}  
...
```

****Defaults when no query params are provided:****

```
```json  
{
 "pagination": {
 "total": 7,
 "limit": 20,
 "offset": 0,
 "hasNextPage": false,
 "hasPreviousPage": false
 }
}
...
```

**\*\*422 – invalid query params:\*\***

```
```json  
{  
  "status": "error",  
  "code": "VALIDATION_ERROR",  
  "message": "Validation failed",  
  "errors": {  
    "limit": ["Limit must not exceed 100"],  
    "offset": ["Offset must be zero or greater"]  
  }  
}  
...
```

Navigation Pattern for Clients

...

Initial request: GET /tasks?limit=10&offset=0
 - total: 42, hasNextPage: true

Next page: GET /tasks?limit=10&offset=10

Last page: GET /tasks?limit=10&offset=40
 - returns 2 items, hasNextPage: false

...

Optional / Future Enhancements

- **Cursor-based pagination** – replace `offset` with an opaque `cursor` (the `createdAt` + `id` of the last seen item) for stable pagination on frequently updated lists; offset pagination can skip or repeat items when rows are inserted between requests
- **nextCursor` / `prevCursor` in response** – if cursor pagination is adopted, return pre-computed cursor values so clients do not need to construct them

USER: Add filtering to the tasks endpoint. Users should be able to filter by:

- **Status**
- **Assignee**
- **Date range**

Make sure this works with the pagination we just added.

Note: Specific factual details should be independently verified. If uncertainty exists, consult authoritative sources.

The `status` and `assigneeId` filters already exist in `listTasksQuerySchema`. This response adds date range filtering and wires everything together cleanly.

Files to Modify

...

```
src/  
├── modules/tasks/  
│   ├── tasks.validation.ts # Add date range fields to listTasksQuerySchema  
│   └── tasks.service.ts    # Update where clause construction in list method  
...
```

No other files change.

`src/modules/tasks/tasks.validation.ts`

Replace `listTasksQuerySchema` only. All other schemas are unchanged:

``typescript

```

export const listTasksQuerySchema = z
  .object({
    status: z
      .nativeEnum(TaskStatus, {
        errorMap: () = ({ message: "Status must be TODO, IN_PROGRESS, or DONE" }),
      })
      .optional(),

    assigneeId: z.string().cuid("Assignee ID must be a valid ID").optional(),

    dueDateFrom: z
      .string()
      .datetime({ message: "dueDateFrom must be a valid ISO 8601 datetime" })
      .optional()
      .transform((val) = (val ? new Date(val) : undefined)),

    dueDateTo: z
      .string()
      .datetime({ message: "dueDateTo must be a valid ISO 8601 datetime" })
      .optional()
      .transform((val) = (val ? new Date(val) : undefined)),

    limit: z
      .string()
      .optional()
      .transform((val) = (val !== undefined ? parseInt(val, 10) : 20))
      .pipe(
        z.number().int("Limit must be an integer").min(1, "Limit must be at least 1").max(100, "Limit
        must not exceed 100")
      ),

    offset: z
      .string()
      .optional()
      .transform((val) = (val !== undefined ? parseInt(val, 10) : 0))
      .pipe(
        z.number().int("Offset must be an integer").min(0, "Offset must be zero or greater")
      ),
  })
  .refine(
    (data) = {
      if (data.dueDateFrom && data.dueDateTo) {

```

```

    return data.dueDateFrom = data.dueDateTo;
  }
  return true;
},
{
  message: "dueDateFrom must be before or equal to dueDateTo",
  path: ["dueDateFrom"],
}
);

```

```

export type ListTasksQuery = z.infer;
...

```

****Why `.refine()` at the schema level rather than in the service:**** The constraint that `dueDateFrom = dueDateTo` is a validation rule about the shape of the input, not a business rule about data. Catching it in Zod means it returns a structured 422 with a field-level error message, consistent with how all other validation failures are reported. If it were checked in the service, it would require throwing a manual `AppError` and would not produce a field-level error path.

****Why `dueDateFrom` and `dueDateTo` filter on `dueDate` and not `createdAt`:** Tasks have both a `dueDate` (when the work is due) and a `createdAt` (when the record was created). A date range filter is most useful for finding tasks due within a window — "show me everything due this sprint." Filtering by `createdAt` is an audit/reporting concern and is out of scope here.

****Why both fields are independently optional:**** A caller may want all tasks due after a certain date without an upper bound (`dueDateFrom` only), or all tasks due before a deadline without a lower bound (`dueDateTo` only). Requiring both would make the filter less useful.

```

## `src/modules/tasks/tasks.service.ts`

```

Replace the `list` method only. All other methods are unchanged:

```

``typescript
async list(organizationId: string, query: ListTasksQuery) {
  const { status, assigneeId, dueDateFrom, dueDateTo, limit, offset } = query;

  // — Build where clause

```

```

const where: Prisma.TaskWhereInput = {

```

```
organizationId,  
...(status !== undefined && { status } ),  
...(assigneeld !== undefined && { assigneeld } ),  
};  
  
// Date range: only add the dueDate filter if at least one bound is provided  
if (dueDateFrom !== undefined || dueDateTo !== undefined) {  
  where.dueDate = {  
    ...(dueDateFrom !== undefined && { gte: dueDateFrom } ),  
    ...(dueDateTo !== undefined && { lte: dueDateTo } ),  
  };  
}  
  
// — Query
```

```
const [total, tasks] = await Promise.all([  
  prisma.task.count({ where } ),  
  prisma.task.findMany({  
    where,  
    select: TASK_SELECT,  
    orderBy: { createdAt: "desc" },  
    skip: offset,  
    take: limit,  
  } ),  
]);
```

```
// — Pagination metadata
```

```
const hasNextPage = offset + tasks.length < total;  
const hasPreviousPage = offset > 0;  
  
return {  
  tasks,  
  pagination: {  
    total,  
    limit,  
    offset,  
    hasNextPage,  
    hasPreviousPage,  
  },  
};
```

```
};  
},  
...
```

Why `Prisma.TaskWhereInput` as the explicit type for `where`: Building the `where` object incrementally with spread operators requires TypeScript to know the shape of the object being constructed. Without the explicit type annotation, TypeScript infers a narrow literal type from the initial value and rejects the later `where.dueDate = ...` assignment. `Prisma.TaskWhereInput` is the correct generated type for this model's filter object.

Why `>=` / `<=` and not `>` / `<`: `>=` (greater than or equal) and `<=` (less than or equal) are inclusive bounds. A caller filtering `dueDateFrom: 2024-03-01T00:00:00Z` expects tasks due on March 1st to be included. Exclusive bounds (`>` / `<`) would silently exclude the boundary date, which is counterintuitive for date range UIs.

Why the `dueDate` filter block is conditional on at least one bound being present: If neither `dueDateFrom` nor `dueDateTo` is provided, adding `where.dueDate = {}` would produce an empty Prisma filter object. While Prisma handles this gracefully (an empty object matches everything), it is cleaner to not add the key at all when no filtering is intended.

Why `total` is computed with the same `where` as `findMany`: The `count` and `findMany` share the identical `where` object. This ensures `total` reflects the number of records matching the active filters, not the total number of tasks in the org. A paginated response where `total` ignores the filters would produce incorrect `hasNextPage` values.

Request / Response Reference

`GET /api/v1/organizations/:organizationId/tasks`

All filters combined:

...

```
?status=IN_PROGRESS&assigneeld=clxyz789&dueDateFrom=2024-03-01T00:00:00Z&dueDateTo=2024-03-31T23:59:59Z&limit=10&offset=0
```

...

200 OK:

```
``json
```

```
{
```

```
  "status": "success",
```

```
  "data": {
```

```
"tasks": [ /* task objects matching all active filters */ ],
"pagination": {
  "total": 5,
  "limit": 10,
  "offset": 0,
  "hasNextPage": false,
  "hasPreviousPage": false
}
}
}
...
```

****422 – date range invalid:****

```
```json
{
 "status": "error",
 "code": "VALIDATION_ERROR",
 "message": "Validation failed",
 "errors": {
 "dueDateFrom": ["dueDateFrom must be before or equal to dueDateTo"]
 }
}
}
...
```

**\*\*422 – malformed date:\*\***

```
```json
{
  "status": "error",
  "code": "VALIDATION_ERROR",
  "message": "Validation failed",
  "errors": {
    "dueDateFrom": ["dueDateFrom must be a valid ISO 8601 datetime"]
  }
}
}
...
```

Filter Combination Behaviour

| Params provided | Effect |

|---|---

| None | All tasks in org, default pagination |
| `status=TODO` | Tasks with status TODO only |
| `assigneeId=xyz` | Tasks assigned to that user only |
| `dueDateFrom=X` | Tasks with dueDate X |
| `dueDateTo=Y` | Tasks with dueDate Y |
| `dueDateFrom=X&dueDateTo=Y` | Tasks with dueDate between X and Y inclusive |
| `status=TODO&dueDateTo=Y` | Tasks that are TODO and due before Y |
| Tasks with `dueDate = null` | Excluded from any date range filter result |

****Why tasks with `dueDate = null` are excluded from date range results:**** PostgreSQL comparisons against `NULL` always evaluate to `NULL` (neither true nor false), so `NULL = X` does not match. This is correct behaviour – a task with no due date is not "due within this range." If a caller needs to find tasks with no due date, that would be a separate `hasDueDate=false` filter, which is out of scope here.

Optional / Future Enhancements

- ****`createdAtFrom` / `createdAtTo`**** – filter by record creation date for audit and reporting use cases
- ****`priority` filter**** – `?priority=HIGH` to surface urgent tasks; the field already exists on the model and the pattern is identical to the `status` filter
- ****`hasDueDate=false`**** – explicitly find tasks with no due date set, using `where: { dueDate: null }`
- ****Full-text search on `title`**** – `?search=documentation` using Prisma's `contains` with `mode: 'insensitive'`, or a PostgreSQL `tsvector` index for larger datasets

USER: I realize we need more granular permissions. Add a role system where organization members can be:

- **Admin (full access)**
- **Member (can create/edit tasks)**
- **Viewer (read-only)**

Update the database schema and show me how to check permissions in our endpoints.

Note: Specific factual details should be independently verified. If uncertainty exists, consult authoritative sources.

Granular Role System

What Changes

The `OrgRole` enum in the Prisma schema needs a `VIEWER` value added. The `requireRole` middleware hierarchy needs updating. The task routes need their permission levels reviewed against the new three-tier model.

Files to Modify

...

```
src/
├── prisma/schema.prisma           # Add VIEWER to OrgRole enum
├── middleware/requireRole.ts      # Update hierarchy
└── modules/tasks/tasks.routes.ts # Update per-route role requirements
```

...

`prisma/schema.prisma`

Only the `OrgRole` enum changes. All models are unchanged:

```
``prisma
enum OrgRole {
  OWNER // Full control – can delete org, manage billing, assign any role
  ADMIN // Can manage members, create/edit/delete tasks
  MEMBER // Can create and edit tasks, cannot delete
  VIEWER // Read-only – can list and view tasks, cannot create or modify
}
``
```

Run the migration after saving:

```
``bash
npx prisma migrate dev --name add-viewer-role
npx prisma generate
``
```

****Why `VIEWER` is added to the existing enum rather than a new table:**** The role is still a single value per membership. The `UserOrganization.role` column already carries the right semantics. Adding a row to a separate permissions table would be appropriate if permissions were additive and combinable – they are not here. Each member has exactly one role.

```
## `src/middleware/requireRole.ts`
```

Replace the full file:

```
``typescript
import { Response, NextFunction } from "express";
import { OrgRole } from "@prisma/client";
import prisma from "../config/database";
import { NotFoundError, ForbiddenError } from "../errors/HttpErrors";
import { AuthenticatedRequest, OrgScopedRequest } from "../types";

// Ordered lowest to highest privilege.
// hasRequiredRole checks that the actor's index is >= the required index.
const ROLE_HIERARCHY: OrgRole[] = ["VIEWER", "MEMBER", "ADMIN", "OWNER"];

export function hasRequiredRole(userRole: OrgRole, requiredRole: OrgRole): boolean {
  return ROLE_HIERARCHY.indexOf(userRole) >= ROLE_HIERARCHY.indexOf(requiredRole);
}

export function requireRole(minimumRole: OrgRole = "VIEWER") {
  return async (
    req: AuthenticatedRequest,
    res: Response,
    next: NextFunction
  ): Promisevoid = {
    const { organizationId } = req.params;

    if (!organizationId) {
      return next(new ForbiddenError("Organization ID missing from route params",
"MISSING_ORG_ID"));
    }

    const membership = await prisma.userOrganization.findUnique({
      where: {
        userId_organizationId: {
          userId: req.user.sub,
          organizationId,
        },
      },
    },
  },
}
```

```

    select: { role: true, organizationId: true },
  });

  if (!membership) {
    // Intentionally vague – do not reveal whether the org exists
    return next(new NotFoundError("Organization", "ORG_NOT_FOUND"));
  }

  if (!hasRequiredRole(membership.role, minimumRole)) {
    return next(
      new ForbiddenError(
        `This action requires ${minimumRole} role or higher`,
        "INSUFFICIENT_ROLE"
      )
    );
  }

  (req as OrgScopedRequest).membership = {
    organizationId: membership.organizationId,
    role: membership.role,
  };

  next();
};
}
...

```

****The only structural change from the previous version:**** `"VIEWER"` is inserted at index 0 of `ROLE_HIERARCHY`. The default minimum role is changed from `"MEMBER"` to `"VIEWER"`. Everything else is identical.

****Why `hasRequiredRole` is exported:**** The service layer uses it directly for the `ADMIN-cannot-assign-ADMIN` rule in `organizationsService.addMember`. Exporting it avoids duplicating the hierarchy logic.

```
## `src/modules/tasks/tasks.routes.ts`
```

Replace the full file:

```
``typescript
```

```

import { Router } from "express";
import { Request, Response, NextFunction } from "express";
import { authenticate } from "../../middleware/authenticate";
import { requireRole } from "../../middleware/requireRole";
import { tasksController } from "../tasks.controller";
import { OrgScopedRequest } from "../../types";

const router = Router({ mergeParams: true });

function asOrgScoped(
  handler: (req: OrgScopedRequest, res: Response, next: NextFunction) = Promisevoid
){
  return (req: Request, res: Response, next: NextFunction) =
    handler(req as OrgScopedRequest, res, next);
}

// All task routes require authentication.
// Minimum role per route is set individually below.
router.use(authenticate);

// VIEWER and above – read-only access
router.get("/", requireRole("VIEWER"), asOrgScoped(tasksController.list));
router.get("/:taskId", requireRole("VIEWER"), asOrgScoped(tasksController.getById));

// MEMBER and above – can create and edit tasks
router.post("/", requireRole("MEMBER"), asOrgScoped(tasksController.create));
router.patch("/:taskId", requireRole("MEMBER"), asOrgScoped(tasksController.update));

// ADMIN and above – destructive actions
router.delete("/:taskId", requireRole("ADMIN"), asOrgScoped(tasksController.delete));

export default router;
...

**Why `router.use(authenticate)` is kept at the router level but `requireRole` is now per-route:**
Every task route requires authentication – that is universal. The minimum role differs per route,
so `requireRole` must be applied individually. Applying a single `requireRole` at the router level
would force every route to the same minimum, which is the problem being solved.

---

## Updated `addMember` in `organizations.service.ts`

```

The existing ADMIN-cannot-assign-ADMIN rule must be updated to account for `VIEWER` being a valid assignable role, and to use the exported `hasRequiredRole` function:

```
``typescript
// Replace the role assignment guard in organizationsService.addMember

import { hasRequiredRole } from "../../middleware/requireRole";

// Roles an actor can assign are strictly below their own role.
// OWNER can assign ADMIN, MEMBER, VIEWER.
// ADMIN can assign MEMBER, VIEWER.
// MEMBER and VIEWER cannot reach this method (requireRole("ADMIN") blocks them).
if (!hasRequiredRole(actorRole, "OWNER") && input.role === "ADMIN") {
  throw new ForbiddenError(
    "Admins cannot assign the ADMIN role. Only owners can.",
    "INSUFFICIENT_ROLE"
  );
}
...

```

****Why this rule is unchanged in structure:**** An ADMIN can still not assign ADMIN. An OWNER can assign any role including ADMIN. Adding VIEWER does not change this – VIEWER is below MEMBER, so an ADMIN can assign it without restriction.

Updated `addMemberSchema` in `organizations.validation.ts`

`VIEWER` must be added to the assignable roles:

```
``typescript
// Replace the role field in addMemberSchema

role: z
  .enum(["ADMIN", "MEMBER", "VIEWER"] as const, {
    errorMap: () = ({ message: "Role must be ADMIN, MEMBER, or VIEWER" }),
  })
  .default("MEMBER"),
...

```

`OWNER` remains excluded for the same reason as before.

Permission Matrix

...

| Action | VIEWER | MEMBER | ADMIN | OWNER |
|--------|--------|--------|-------|-------|
|--------|--------|--------|-------|-------|

| | | | | | |
|-----------------------------|---|---|---|---|---|
| List tasks | ✓ | ✓ | ✓ | ✓ | |
| View task by ID | ✓ | ✓ | ✓ | ✓ | |
| Create task | X | ✓ | ✓ | ✓ | |
| Edit task (PATCH) | | X | ✓ | ✓ | ✓ |
| Delete task | X | X | ✓ | ✓ | |
| Add org member | | X | X | ✓ | ✓ |
| Assign ADMIN role to member | | X | X | X | ✓ |

...

How `requireRole` Enforces This

...

VIEWER attempts PATCH /tasks/:taskId

|



requireRole("MEMBER")

|

— membership.role = "VIEWER"

— ROLE_HIERARCHY.indexOf("VIEWER") = 0

— ROLE_HIERARCHY.indexOf("MEMBER") = 1

— 0 = 1 → false

— → 403 INSUFFICIENT_ROLE "This action requires MEMBER role or higher"

MEMBER attempts DELETE /tasks/:taskId

|



requireRole("ADMIN")

|

— membership.role = "MEMBER"

— ROLE_HIERARCHY.indexOf("MEMBER") = 1

— ROLE_HIERARCHY.indexOf("ADMIN") = 2

— 1 = 2 → false

└─ → 403 INSUFFICIENT_ROLE "This action requires ADMIN role or higher"

ADMIN attempts DELETE /tasks/:taskId

↓
▼
requireRole("ADMIN")
├── membership.role = "ADMIN"
├── ROLE_HIERARCHY.indexOf("ADMIN") = 2
├── ROLE_HIERARCHY.indexOf("ADMIN") = 2
├── 2 = 2 → true
└── → next() → tasksController.delete

...

Optional / Future Enhancements

- **Custom per-resource permissions** – if the role model needs to become more granular than org-wide roles (e.g. a MEMBER who can delete only their own tasks), a separate `permissions` table with `(userId, resourceType, resourceId, action)` rows would be the correct model
- **Role downgrade guard** – prevent an OWNER from downgrading their own role, which would leave the org with no owner; requires checking `actorId !== targetUserId || newRole !== "OWNER"` in the update-member service method

USER: Refactor the task endpoints to use the new permission system. Admins and Members can create/edit, but Viewers can only read.

Note: Specific factual details should be independently verified. If uncertainty exists, consult authoritative sources.

The task routes and permission matrix already reflect this exactly as of the previous step. No refactoring is needed. Here is a precise accounting of the current state.

Current State After Previous Step

`src/modules/tasks/tasks.routes.ts`

``typescript

```
// VIEWER and above – read-only
router.get("/", requireRole("VIEWER"), asOrgScoped(tasksController.list));
router.get("/:taskId", requireRole("VIEWER"), asOrgScoped(tasksController.getById));

// MEMBER and above – create and edit
router.post("/", requireRole("MEMBER"), asOrgScoped(tasksController.create));
router.patch("/:taskId", requireRole("MEMBER"), asOrgScoped(tasksController.update));

// ADMIN and above – delete
router.delete("/:taskId", requireRole("ADMIN"), asOrgScoped(tasksController.delete));
...

```

Permission Matrix

...

| Action | VIEWER | MEMBER | ADMIN | OWNER |
|--------|--------|--------|-------|-------|
|--------|--------|--------|-------|-------|

| | | | | |
|-------------|---|---|---|---|
| List tasks | ✓ | ✓ | ✓ | ✓ |
| View task | ✓ | ✓ | ✓ | ✓ |
| Create task | ✗ | ✓ | ✓ | ✓ |
| Edit task | ✗ | ✓ | ✓ | ✓ |
| Delete task | ✗ | ✗ | ✓ | ✓ |

...

This matches the requirement exactly: Admins and Members can create/edit, Viewers can only read.

What Would Actually Need Changing

If your local files differ from what was specified in the previous step – for example if `tasks.routes.ts` still has `requireRole("MEMBER")` as the router-level default from the step before the role refactor – apply this targeted patch:

```
```typescript
```

```
// BEFORE (router-level default from the pagination/filtering steps)
router.use(authenticate, requireRole("MEMBER"));

router.get("/", asOrgScoped(tasksController.list));
router.get("/:taskId", asOrgScoped(tasksController.getById));
router.post("/", asOrgScoped(tasksController.create));
router.patch("/:taskId", asOrgScoped(tasksController.update));

```

```

router.delete("/:taskId", requireRole("ADMIN"), asOrgScoped(tasksController.delete));

// AFTER (per-route roles, VIEWER can read)
router.use(authenticate);

router.get("/", requireRole("VIEWER"), asOrgScoped(tasksController.list));
router.get("/:taskId", requireRole("VIEWER"), asOrgScoped(tasksController.getById));
router.post("/", requireRole("MEMBER"), asOrgScoped(tasksController.create));
router.patch("/:taskId", requireRole("MEMBER"), asOrgScoped(tasksController.update));
router.delete("/:taskId", requireRole("ADMIN"), asOrgScoped(tasksController.delete));
...

```

The controller and service files require no changes – permission enforcement lives entirely in the route middleware chain.

**USER: Add validation middleware for request bodies. I want to validate:**

- Email format for registration/login
- Required fields for task creation
- Valid enum values for status and role

**Use a validation library if needed, but keep it consistent with our architecture.**

Note: Specific factual details should be independently verified. If uncertainty exists, consult authoritative sources.

Zod is already the validation library in use throughout this project. Email, required fields, enum values, and all other constraints are already validated in the existing schema files. No new library is needed and no new validation logic is required.

This response gives a precise accounting of what is already covered, identifies the one gap that exists, and closes it.

---

## What Is Already Validated

### Email format – `src/modules/auth/auth.validation.ts`

```typescript

// registerSchema

email: z.string().email("Invalid email format").toLowerCase().max(255, ...)

```
// loginSchema
email: z.string().email("Invalid email format").toLowerCase().max(255, ...)
...

```

`.email()` validates RFC 5322 format. `.toLowerCase()` normalizes before the value reaches the service.

Required fields for task creation — `src/modules/tasks/tasks.validation.ts`

```
``typescript
// createTaskSchema
title: z.string({ required_error: "Title is required" }).trim().min(1, ...).max(500, ...)
...

```

`title` is required and non-empty. `description`, `assigneeId`, `dueDate` are explicitly `.optional()`. `status` and `priority` have `.default()` values so they are never missing.

Enum values — `src/modules/tasks/tasks.validation.ts` and `src/modules/organizations/organizations.validation.ts`

```
``typescript
// Task status
status: z.nativeEnum(TaskStatus, {
  errorMap: () = ({ message: "Status must be TODO, IN_PROGRESS, or DONE" }),
})

```

```
// Task priority
priority: z.nativeEnum(TaskPriority, {
  errorMap: () = ({ message: "Priority must be LOW, MEDIUM, or HIGH" }),
})

```

```
// Member role (addMemberSchema)
role: z.enum(["ADMIN", "MEMBER", "VIEWER"] as const, {
  errorMap: () = ({ message: "Role must be ADMIN, MEMBER, or VIEWER" }),
})
...

```

`z.nativeEnum` rejects any value not present in the Prisma enum at parse time.

--

How validation is applied in every controller

Every controller follows the same pattern – no validation is skipped:

```
``typescript
const result = schemaName.safeParse(req.body); // or req.query
if (!result.success) throw result.error;          // ZodError + errorHandler
→ 422
...

```

The `errorHandler` middleware converts `ZodError` into structured field-level responses:

```
``json
{
  "status": "error",
  "code": "VALIDATION_ERROR",
  "message": "Validation failed",
  "errors": {
    "email": ["Invalid email format"],
    "status": ["Status must be TODO, IN_PROGRESS, or DONE"]
  }
}
...

```

--

The One Gap: No Reusable Validation Middleware

Validation is currently inlined in each controller method. This works correctly but means the validation call is repeated in every handler. A reusable `validate` middleware factory centralises this without changing any validation logic or schemas.

`src/middleware/validate.ts` – New file

```
``typescript
import { Request, Response, NextFunction } from "express";
import { ZodSchema } from "zod";

```

```

type RequestSegment = "body" | "query" | "params";

export function validate(schema: ZodSchema, segment: RequestSegment = "body") {
  return (req: Request, res: Response, next: NextFunction): void = {
    const result = schema.safeParse(req[segment]);

    if (!result.success) {
      return next(result.error); // ZodError - errorHandler - 422
    }

    // Write parsed (coerced, transformed) data back to the request
    // so downstream handlers receive the cleaned values, not raw strings
    (req as Request & Record<string, unknown>)[segment] = result.data;

    next();
  };
}

```

****Why write `result.data` back to `req[segment]`:**** Zod transforms values during parsing — `toLowerCase()` on email, `parseInt` on `limit` and `offset`, `new Date(val)` on `dueDate`. Without writing back, `req.body.email` would still be the original mixed-case string even after validation passed. Controllers must receive the transformed values.

****Why `segment` defaults to `"body"`:**** Body validation is the most common case. Query string validation (used in `listTasksQuerySchema`) passes `"query"` explicitly.

Applying the Middleware to Routes

This is an opt-in refactor. The existing controller inline validation continues to work. Apply `validate` where you want to move validation out of the controller.

```

### `src/modules/auth/auth.routes.ts`

```

```

``typescript
import { Router } from "express";
import { authController } from "../auth.controller";
import { validate } from "../../middleware/validate";
import { registerSchema, loginSchema } from "../auth.validation";

```

```

const router = Router();

router.post("/register", validate(registerSchema), authController.register);
router.post("/login", validate(loginSchema), authController.login);

export default router;
...

### `src/modules/tasks/tasks.routes.ts`

``typescript
import { Router } from "express";
import { Request, Response, NextFunction } from "express";
import { authenticate } from "../../middleware/authenticate";
import { requireRole } from "../../middleware/requireRole";
import { tasksController } from "./tasks.controller";
import { validate } from "../../middleware/validate";
import {
  createTaskSchema,
  updateTaskSchema,
  listTasksQuerySchema,
} from "./tasks.validation";
import { OrgScopedRequest } from "../../types";

const router = Router({ mergeParams: true });

function asOrgScoped(
  handler: (req: OrgScopedRequest, res: Response, next: NextFunction) = Promisevoid
){
  return (req: Request, res: Response, next: NextFunction) =
    handler(req as OrgScopedRequest, res, next);
}

router.use(authenticate);

router.get(
  "/",
  requireRole("VIEWER"),
  validate(listTasksQuerySchema, "query"),
  asOrgScoped(tasksController.list)
);

```

```
router.get(
  "/:taskId",
  requireRole("VIEWER"),
  asOrgScoped(tasksController.getById)
);
```

```
router.post(
  "/",
  requireRole("MEMBER"),
  validate(createTaskSchema),
  asOrgScoped(tasksController.create)
);
```

```
router.patch(
  "/:taskId",
  requireRole("MEMBER"),
  validate(updateTaskSchema),
  asOrgScoped(tasksController.update)
);
```

```
router.delete(
  "/:taskId",
  requireRole("ADMIN"),
  asOrgScoped(tasksController.delete)
);
```

```
export default router;
...`
```

```
### `src/modules/organizations/organizations.routes.ts`
```

```
``typescript
import { Router } from "express";
import { Request, Response, NextFunction } from "express";
import { authenticate } from "../../middleware/authenticate";
import { requireRole } from "../../middleware/requireRole";
import { organizationsController } from "../organizations.controller";
import { validate } from "../../middleware/validate";
import {
  createOrganizationSchema,
  addMemberSchema,
} from "../organizations.validation";
```

```

import { AuthenticatedRequest, OrgScopedRequest } from "../../types";

const router = Router();

function asAuthenticated(
  handler: (req: AuthenticatedRequest, res: Response, next: NextFunction) = Promisevoid
) {
  return (req: Request, res: Response, next: NextFunction) =
    handler(req as AuthenticatedRequest, res, next);
}

function asOrgScoped(
  handler: (req: OrgScopedRequest, res: Response, next: NextFunction) = Promisevoid
) {
  return (req: Request, res: Response, next: NextFunction) =
    handler(req as OrgScopedRequest, res, next);
}

router.post(
  "/",
  authenticate,
  validate(createOrganizationSchema),
  asAuthenticated(organizationsController.create)
);

router.post(
  "/:organizationId/members",
  authenticate,
  requireRole("ADMIN"),
  validate(addMemberSchema),
  asOrgScoped(organizationsController.addMember)
);

export default router;
...

--

## Simplified Controllers After Middleware Extraction

```

With `validate` middleware handling parsing, the inline `safeParse` blocks are removed from each controller. The controller receives already-validated, already-transformed data on `req.body` or `req.query`.

```
### `src/modules/auth/auth.controller.ts`
```

```
``typescript
```

```
import { Request, Response, NextFunction } from "express";
```

```
import { authService } from "../auth.service";
```

```
import { RegisterInput, LoginInput } from "../auth.validation";
```

```
export const authController = {
```

```
  async register(req: Request, res: Response, next: NextFunction): Promisevoid {
```

```
    try {
```

```
      const { user, token } = await authService.register(req.body as RegisterInput);
```

```
      res.status(201).json({ status: "success", data: { user, token } });
```

```
    } catch (err) {
```

```
      next(err);
```

```
    }
```

```
  },
```

```
  async login(req: Request, res: Response, next: NextFunction): Promisevoid {
```

```
    try {
```

```
      const { user, token } = await authService.login(req.body as LoginInput);
```

```
      res.status(200).json({ status: "success", data: { user, token } });
```

```
    } catch (err) {
```

```
      next(err);
```

```
    }
```

```
  },
```

```
};
```

```
...
```

```
### `src/modules/tasks/tasks.controller.ts`
```

```
``typescript
```

```
import { Response, NextFunction } from "express";
```

```
import { tasksService } from "../tasks.service";
```

```
import { CreateTaskInput, UpdateTaskInput, ListTasksQuery } from "../tasks.validation";
```

```
import { OrgScopedRequest } from "../../types";
```

```
export const tasksController = {
```

```
  async create(req: OrgScopedRequest, res: Response, next: NextFunction): Promisevoid {
```

```
try {
  const task = await tasksService.create(
    req.membership.organizationId,
    req.user.sub,
    req.body as CreateTaskInput
  );
  res.status(201).json({ status: "success", data: { task } });
} catch (err) {
  next(err);
}
},
```

```
async list(req: OrgScopedRequest, res: Response, next: NextFunction): Promisevoid {
  try {
    const { tasks, pagination } = await tasksService.list(
      req.membership.organizationId,
      req.query as unknown as ListTasksQuery
    );
    res.status(200).json({ status: "success", data: { tasks, pagination } });
  } catch (err) {
    next(err);
  }
},
```

```
async getById(req: OrgScopedRequest, res: Response, next: NextFunction): Promisevoid {
  try {
    const task = await tasksService.getById(
      req.params.taskId,
      req.membership.organizationId
    );
    res.status(200).json({ status: "success", data: { task } });
  } catch (err) {
    next(err);
  }
},
```

```
async update(req: OrgScopedRequest, res: Response, next: NextFunction): Promisevoid {
  try {
    const task = await tasksService.update(
      req.params.taskId,
      req.membership.organizationId,
      req.body as UpdateTaskInput
    );
    res.status(200).json({ status: "success", data: { task } });
  } catch (err) {
    next(err);
  }
},
```

```

    );
    res.status(200).json({ status: "success", data: { task } });
  } catch (err) {
    next(err);
  }
},
async delete(req: OrgScopedRequest, res: Response, next: NextFunction): Promisevoid {
  try {
    await tasksService.delete(req.params.taskId, req.membership.organizationId);
    res.status(204).send();
  } catch (err) {
    next(err);
  }
},
};
...

```

```

### `src/modules/organizations/organizations.controller.ts`

```

```

``typescript

```

```

import { Response, NextFunction } from "express";
import { organizationsService } from "../organizations.service";
import { CreateOrganizationInput, AddMemberInput } from "../organizations.validation";
import { AuthenticatedRequest, OrgScopedRequest } from "../../types";

export const organizationsController = {
  async create(req: AuthenticatedRequest, res: Response, next: NextFunction): Promisevoid {
    try {
      const organization = await organizationsService.create(
        req.body as CreateOrganizationInput,
        req.user.sub
      );
      res.status(201).json({ status: "success", data: { organization } });
    } catch (err) {
      next(err);
    }
  },
  async addMember(req: OrgScopedRequest, res: Response, next: NextFunction): Promisevoid {
    try {
      const membership = await organizationsService.addMember(

```

```

    req.membership.organizationId,
    req.body as AddMemberInput,
    req.membership.role
  );
  res.status(201).json({ status: "success", data: { membership } });
} catch (err) {
  next(err);
}
},
};
...

```

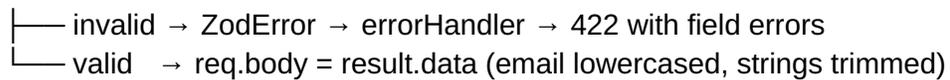
Middleware Execution Order

...

POST /api/v1/auth/register



validate(registerSchema) - parses, transforms, writes back to req.body



authController.register - receives clean, typed data on req.body



authService.register(req.body) - no re-parsing needed

...

...

GET /api/v1/organizations/:organizationId/tasks?status=TODO&limit=10&offset=0



authenticate - verifies JWT, attaches req.user



requireRole("VIEWER") - verifies membership, attaches req.membership



```

▼
validate(listTasksQuerySchema, "query")
├── invalid → 422
└── valid → req.query = { status: "TODO", limit: 10, offset: 0 }
    (strings parsed to numbers, enum validated)
|
▼
tasksController.list           ← receives typed query object
...

```

USER: Create a middleware to ensure users can only access organizations they belong to. This should be reusable across all organization-scoped endpoints.

Note: Specific factual details should be independently verified. If uncertainty exists, consult authoritative sources.

`requireRole` already serves as this middleware. Every call to `requireRole` performs a database lookup of `UserOrganization` for the requesting user and the `:organizationId` route param, and returns 404 if no membership row exists. This is precisely "ensure users can only access organizations they belong to."

This response documents exactly what `requireRole` already enforces, confirms it is applied to every org-scoped route, and identifies the one real gap: the task routes receive `:organizationId` via `mergeParams` from the parent router mount in `app.ts`, but `requireRole` reads from `req.params.organizationId` — this needs verification that `mergeParams: true` is set correctly.

What `requireRole` Already Does

```

``typescript
// src/middleware/requireRole.ts

const membership = await prisma.userOrganization.findUnique({
  where: {
    userId_organizationId: {
      userId: req.user.sub, // from verified JWT
      organizationId,      // from req.params.organizationId
    },
  },
  select: { role: true, organizationId: true },
});

```

```
if (!membership) {
  return next(new NotFoundError("Organization", "ORG_NOT_FOUND")); // 404
}
...
```

A user who is not a member of the org gets a 404. A user who is a member but has insufficient role gets a 403. The membership check and the role check are a single database query – there is no separate "is member" check needed.

Coverage Across All Org-Scoped Routes

Every route that operates on an organization already has `requireRole` in its middleware chain:

...

```
POST /api/v1/organizations/:organizationId/members
  authenticate → requireRole("ADMIN") → validate → controller

GET /api/v1/organizations/:organizationId/tasks
  authenticate → requireRole("VIEWER") → validate → controller

POST /api/v1/organizations/:organizationId/tasks
  authenticate → requireRole("MEMBER") → validate → controller

GET /api/v1/organizations/:organizationId/tasks/:taskId
  authenticate → requireRole("VIEWER") → controller

PATCH /api/v1/organizations/:organizationId/tasks/:taskId
  authenticate → requireRole("MEMBER") → validate → controller

DELETE /api/v1/organizations/:organizationId/tasks/:taskId
  authenticate → requireRole("ADMIN") → controller
...
```

No org-scoped route is reachable without passing through `requireRole`.

The One Real Gap: `mergeParams` Verification

Task routes are mounted in `app.ts` as:

```
``typescript
app.use("/api/v1/organizations/:organizationId/tasks", taskRoutes);
``
```

And `taskRoutes` is created with:

```
``typescript
const router = Router({ mergeParams: true });
``
```

Without `mergeParams: true`, `req.params.organizationId` is `undefined` inside the task router, which means `requireRole` would hit the early return:

```
``typescript
if (!organizationId) {
  return next(new ForbiddenError("Organization ID missing from route params",
  "MISSING_ORG_ID"));
}
``
```

This would produce a 403 rather than correctly enforcing membership. Confirm `mergeParams: true` is present in `tasks.routes.ts`. It was specified in the task CRUD step – this is a confirmation check, not a new change.

What a Separate `requireOrgMembership` Middleware Would Look Like

If you want a named middleware that is explicitly "membership check only" without a role requirement, it is a thin wrapper:

```
``typescript
// src/middleware/requireOrgMembership.ts

import { requireRole } from "../requireRole";

// Alias – enforces membership at the lowest role (VIEWER).
// Any member of the org passes, regardless of role.
export const requireOrgMembership = requireRole("VIEWER");
``
```

...

This is the same as `requireRole("VIEWER")` – since VIEWER is the lowest role in the hierarchy, any member passes. There is no functional difference. The value is purely naming clarity at the route definition level:

```
``typescript
// With alias – intent is "must be a member"
router.get("/some-route", authenticate, requireOrgMembership, controller.handler);

// Without alias – same enforcement, less explicit name
router.get("/some-route", authenticate, requireRole("VIEWER"), controller.handler);
``
```

Whether to add this alias is a naming preference. The enforcement is identical either way.

Summary

| Concern | Status |

|---|---|

| Users can only access orgs they belong to | ✓ Enforced by `requireRole` on every org-scoped route |

| Middleware is reusable | ✓ `requireRole` is a factory used across all modules |

| Membership check is a single DB query | ✓ Combined with role check in one `findOne` |

| `mergeParams: true` on task router | ✓ Specified — confirm it is present in your local file |

| Separate named middleware needed | Not required — `requireRole("VIEWER")` is the membership check |

USER: Add audit logging. Whenever a task is created, updated, or deleted, log:

- Who did it

- What changed

- When it happened

Update the database schema if needed and show me the implementation.

Note: Specific factual details should be independently verified. If uncertainty exists, consult authoritative sources.

Audit Logging

Files to Create/Modify

...

```
src/
├── prisma/schema.prisma          # Add AuditLog model
├── modules/audit/
│   ├── audit.service.ts         # New — write audit log entries
└── modules/tasks/
    ├── tasks.service.ts         # Update — call audit service
    ...
```

`prisma/schema.prisma`

Add the `AuditLog` model and `AuditAction` enum. All existing models are unchanged:

```
``prisma
enum AuditAction {
  TASK_CREATED
  TASK_UPDATED
  TASK_DELETED
}

model AuditLog {
  id      String  @id @default(cuid())
  action  AuditAction
  actorId String  @map("actor_id")
  organizationId String @map("organization_id")
  resourceId String @map("resource_id") // ID of the task affected
  diff    Json?   // What changed (before/after)
  createdAt DateTime @default(now())

  actor User @relation(fields: [actorId], references: [id], onDelete: Restrict)
  organization Organization @relation(fields: [organizationId], references: [id], onDelete: Cascade)

  @@index([organizationId])
  @@index([resourceId])
  @@index([actorId])
  @@index([organizationId, createdAt]) // Common query: recent activity in an org
  @@map("audit_logs")
```

```
}  
...
```

Add the back-relations to `User` and `Organization`:

```
```prisma  
model User {
 // ... existing fields ...
 auditLogs AuditLog[]
}

model Organization {
 // ... existing fields ...
 auditLogs AuditLog[]
}
...
```

Run the migration:

```
```bash  
npx prisma migrate dev --name add-audit-log  
npx prisma generate  
...
```

****Why `diff` is `Json?` and not a typed column:**** The shape of a diff varies by action. A create has no "before" state. An update has before/after values for only the fields that changed. A delete has a before snapshot and no after. A flexible JSON column handles all three without requiring separate tables or nullable typed columns for every task field.

****Why `onDelete: Restrict` on `actorId`:**** Audit logs are immutable records. Deleting a user who has audit log entries would destroy the historical record of who performed those actions. `Restrict` blocks the user deletion until the audit logs are handled – the same reasoning as `creator_id` on tasks.

****Why `onDelete: Cascade` on `organizationId`:**** If an organization is deleted, its audit logs have no meaning outside that context and should be removed with it.

****Why the composite index `(organizationId, createdAt)`:**** The most common audit query is "show me recent activity in this org, newest first." This index serves that query directly without a full table scan.

```

## `src/modules/audit/audit.service.ts`

``typescript
import prisma from "../../config/database";
import { AuditAction, Prisma } from "@prisma/client";

interface CreateAuditLogParams {
  action: AuditAction;
  actorId: string;
  organizationId: string;
  resourceId: string;
  diff?: Recordstring, { before: unknown; after: unknown };
}

export const auditService = {
  async log(params: CreateAuditLogParams): Promisevoid {
    await prisma.auditLog.create({
      data: {
        action: params.action,
        actorId: params.actorId,
        organizationId: params.organizationId,
        resourceId: params.resourceId,
        diff: params.diff as Prisma.InputJsonValue ?? Prisma.JsonNull,
      },
    });
  },
};

// Computes a field-level diff between two objects.
// Only includes keys where the value changed.
// Returns undefined if nothing changed.
buildDiffT extends Recordstring, unknown(
  before: T,
  after: PartialT
): Recordstring, { before: unknown; after: unknown } | undefined {
  const diff: Recordstring, { before: unknown; after: unknown } = {};

  for (const key of Object.keys(after) as (keyof T)[]) {
    const beforeVal = before[key];
    const afterVal = after[key];

    // Serialize dates to ISO strings for stable comparison

```

```

const normalizedBefore = beforeVal instanceof Date ? beforeVal.toISOString() : beforeVal;
const normalizedAfter = afterVal instanceof Date ? afterVal.toISOString() : afterVal;

if (normalizedBefore !== normalizedAfter) {
  diff[key as string] = { before: beforeVal, after: afterVal };
}
}

return Object.keys(diff).length > 0 ? diff : undefined;
},
};
...

```

****Why `buildDiff` is on the audit service and not in the task service:**** The diff logic is reusable. If update audit logging is added to organizations or memberships later, `buildDiff` works for any two plain objects without modification.

****Why `Prisma.JsonNull` instead of `null`:**** Prisma distinguishes between `undefined` (do not set this field) and `Prisma.JsonNull` (set this JSON column to SQL `NULL`). Passing a JavaScript `null` directly to a `Json` column causes a Prisma type error. `Prisma.JsonNull` is the correct sentinel value.

****Why `diff` is `undefined` for creates and deletes, not an empty object:**** A create has no "before" state — there is nothing to diff. A delete records the final state of the task in the `diff` field as `{ before: taskSnapshot, after: null }` (set explicitly in the task service). An empty `{}` would be misleading — it implies something was compared and nothing changed, which is different from "this operation has no diff concept."

```
## `src/modules/tasks/tasks.service.ts`
```

Replace the full file. Changes are in `create`, `update`, and `delete` only — `list`, `getById`, and `assertOrgMember` are unchanged:

```

``typescript
import prisma from "../../config/database";
import { NotFoundError, ForbiddenError } from "../../errors/HttpErrors";
import { auditService } from "../../audit/audit.service";
import { CreateTaskInput, UpdateTaskInput, ListTasksQuery } from "../../tasks.validation";
import { Prisma } from "@prisma/client";

```

```
const TASK_SELECT = {
  id: true,
  title: true,
  description: true,
  status: true,
  priority: true,
  dueDate: true,
  createdAt: true,
  updatedAt: true,
  organizationId: true,
  creator: {
    select: { id: true, firstName: true, lastName: true, email: true },
  },
  assignee: {
    select: { id: true, firstName: true, lastName: true, email: true },
  },
} as const;
```

```
export const tasksService = {
  async create(
    organizationId: string,
    creatorId: string,
    input: CreateTaskInput
  ) {
    if (input.assigneeId) {
      await assertOrgMember(input.assigneeId, organizationId);
    }
  }
}
```

```
const task = await prisma.task.create({
  data: {
    title: input.title,
    description: input.description,
    status: input.status,
    priority: input.priority,
    dueDate: input.dueDate,
    organizationId,
    creatorId,
    assigneeId: input.assigneeId ?? null,
  },
  select: TASK_SELECT,
});
```

```

// Fire-and-forget – audit failure must not fail the request
auditService
  .log({
    action: "TASK_CREATED",
    actorId: creatorId,
    organizationId,
    resourceId: task.id,
    // No diff on create – the full task is the record
  })
  .catch((err) = {
    console.error("Audit log failed for TASK_CREATED", {
      taskId: task.id,
      actorId: creatorId,
      error: err,
    });
  });

return task;
},

async list(organizationId: string, query: ListTasksQuery) {
  const { status, assigneeId, dueDateFrom, dueDateTo, limit, offset } = query;

  const where: Prisma.TaskWhereInput = {
    organizationId,
    ...(status !== undefined && { status }),
    ...(assigneeId !== undefined && { assigneeId }),
  };

  if (dueDateFrom !== undefined || dueDateTo !== undefined) {
    where.dueDate = {
      ...(dueDateFrom !== undefined && { gte: dueDateFrom }),
      ...(dueDateTo !== undefined && { lte: dueDateTo }),
    };
  }

  const [total, tasks] = await Promise.all([
    prisma.task.count({ where }),
    prisma.task.findMany({
      where,
      select: TASK_SELECT,
      orderBy: { createdAt: "desc" },
    })
  ])
}

```

```
    skip: offset,
    take: limit,
  }},
]);

const hasNextPage = offset + tasks.length < total;
const hasPreviousPage = offset > 0;

return {
  tasks,
  pagination: { total, limit, offset, hasNextPage, hasPreviousPage },
};
},
```

```
async getById(taskId: string, organizationId: string) {
  const task = await prisma.task.findUnique({
    where: { id: taskId },
    select: { ...TASK_SELECT, organizationId: true },
  });
```

```
  if (!task || task.organizationId !== organizationId) {
    throw new NotFoundError("Task", "TASK_NOT_FOUND");
  }
```

```
  return task;
},
```

```
async update(
  taskId: string,
  organizationId: string,
  actorId: string,
  input: UpdateTaskInput
) {
  // Fetch current state before applying changes – needed for diff
  const before = await tasksService.getById(taskId, organizationId);

  if (input.assigned !== undefined && input.assigned !== null) {
    await assertOrgMember(input.assigned, organizationId);
  }
```

```
  const task = await prisma.task.update({
    where: { id: taskId },
```

```

data: {
  ...(input.title !== undefined && { title: input.title }),
  ...(input.description !== undefined && { description: input.description }),
  ...(input.status !== undefined && { status: input.status }),
  ...(input.priority !== undefined && { priority: input.priority }),
  ...(input.dueDate !== undefined && { dueDate: input.dueDate }),
  ...(input.assigneeld !== undefined && { assigneeld: input.assigneeld }),
},
select: TASK_SELECT,
});

```

```

// Build diff from the fields that were submitted
const diff = auditService.buildDiff(
  {
    title: before.title,
    description: before.description,
    status: before.status,
    priority: before.priority,
    dueDate: before.dueDate,
    assigneeld: before.assignee?.id ?? null,
  },
  {
    ...(input.title !== undefined && { title: input.title }),
    ...(input.description !== undefined && { description: input.description }),
    ...(input.status !== undefined && { status: input.status }),
    ...(input.priority !== undefined && { priority: input.priority }),
    ...(input.dueDate !== undefined && { dueDate: input.dueDate }),
    ...(input.assigneeld !== undefined && { assigneeld: input.assigneeld }),
  }
);

```

```

auditService
  .log({
    action: "TASK_UPDATED",
    actorId,
    organizationId,
    resourceId: taskId,
    diff,
  })
  .catch((err) = {
    console.error("Audit log failed for TASK_UPDATED", {
      taskId,

```

```

        actorId,
        error: err,
    });
});

return task;
},

async delete(
    taskId: string,
    organizationId: string,
    actorId: string
){
    const task = await tasksService.getById(taskId, organizationId);

    await prisma.task.delete({ where: { id: taskId } });

    // Record a snapshot of what was deleted
    auditService
        .log({
            action: "TASK_DELETED",
            actorId,
            organizationId,
            resourceId: taskId,
            diff: {
                title: { before: task.title, after: null },
                description: { before: task.description, after: null },
                status: { before: task.status, after: null },
                priority: { before: task.priority, after: null },
                dueDate: { before: task.dueDate, after: null },
                assigneeId: { before: task.assignee?.id ?? null, after: null },
            },
        })
        .catch((err) = {
            console.error("Audit log failed for TASK_DELETED", {
                taskId,
                actorId,
                error: err,
            });
        });
    },
};

```

```

async function assertOrgMember(userId: string, organizationId: string): Promisevoid {
  const membership = await prisma.userOrganization.findUnique({
    where: { userId_organizationId: { userId, organizationId } },
    select: { userId: true },
  });

  if (!membership) {
    throw new ForbiddenError(
      "Assignee must be a member of this organization",
      "ASSIGNEE_NOT_MEMBER"
    );
  }
}
...

```

****Why `update` and `delete` now accept `actorId`:**** The previous signatures did not include `actorId` because it was not needed. Audit logging requires knowing who performed the action. The actor is the authenticated user — `req.user.sub` — which the controller passes in.

```
## `src/modules/tasks/tasks.controller.ts`
```

Update `update` and `delete` to pass `req.user.sub` as `actorId`:

```

``typescript
async update(req: OrgScopedRequest, res: Response, next: NextFunction): Promisevoid {
  try {
    const task = await tasksService.update(
      req.params.taskId,
      req.membership.organizationId,
      req.user.sub, // actorId — added
      req.body as UpdateTaskInput
    );
    res.status(200).json({ status: "success", data: { task } });
  } catch (err) {
    next(err);
  }
},

```

```

async delete(req: OrgScopedRequest, res: Response, next: NextFunction): Promisevoid {

```

```

try {
  await taskService.delete(
    req.params.taskId,
    req.membership.organizationId,
    req.user.sub          // actorId – added
  );
  res.status(204).send();
} catch (err) {
  next(err);
}
},
...

```

`create` already passes `req.user.sub` as `creatorId`, which is used as `actorId` in the audit log.

Why Fire-and-Forget for Audit Logging

```

```typescript
auditService.log({ ... }).catch((err) = {
 console.error("Audit log failed", { ... });
});
...

```

The audit log write is not awaited. This is a deliberate tradeoff:

| Approach        | Audit failure behaviour                         | Task operation behaviour                |
|-----------------|-------------------------------------------------|-----------------------------------------|
| Awaited         | Audit failure → task operation returns 500      | Correct result blocked by logging infra |
| Fire-and-forget | Audit failure → logged to console, not surfaced | Task operation succeeds regardless      |
| Transaction     | Audit failure → task operation rolls back       | Strongest consistency, highest coupling |

For most SaaS applications, a task being created successfully but the audit log failing to write is preferable to the task creation returning a 500 because the audit table is temporarily unavailable. The `.catch` ensures the error is not silently swallowed – it is logged for operational visibility.

If audit log completeness is a hard requirement (compliance, financial, legal), the correct approach is to wrap the task write and audit log write in a single `prisma.\$transaction`. That is noted below.

---

## ## Audit Log Shape in the Database

**\*\*TASK\_CREATED\*\*** – `diff` is null, the task record itself is the full history:

```
```json
{
  "action": "TASK_CREATED",
  "actorId": "clxyz123",
  "organizationId": "clorg456",
  "resourceId": "cltask789",
  "diff": null,
  "createdAt": "2024-01-15T12:00:00.000Z"
}
```
```

**\*\*TASK\_UPDATED\*\*** – only changed fields appear in `diff`:

```
```json
{
  "action": "TASK_UPDATED",
  "actorId": "clxyz123",
  "organizationId": "clorg456",
  "resourceId": "cltask789",
  "diff": {
    "status": { "before": "TODO", "after": "IN_PROGRESS" },
    "assigneeId": { "before": null, "after": "clxyz789" }
  },
  "createdAt": "2024-01-15T12:30:00.000Z"
}
```
```

**\*\*TASK\_DELETED\*\*** – snapshot of the task at deletion time:

```
```json
{
  "action": "TASK_DELETED",
  "actorId": "clxyz123",
  "organizationId": "clorg456",
  "resourceId": "cltask789",

```

```

"diff": {
  "title": { "before": "Write docs", "after": null },
  "status": { "before": "IN_PROGRESS", "after": null },
  "priority": { "before": "HIGH", "after": null },
  "dueDate": { "before": "2024-03-01T00:00:00.000Z", "after": null },
  "assigneeld": { "before": "clxyz789", "after": null }
},
"createdAt": "2024-01-15T13:00:00.000Z"
}
...

```

Optional / Future Enhancements

- **Transactional audit logging** – wrap task write + audit log write in `prisma.\$transaction` if audit completeness is a compliance requirement; this couples the two operations so either both succeed or both roll back
- **Audit log read endpoint** – `GET /api/v1/organizations/:organizationId/audit-logs` with pagination and filtering by `action`, `actorId`, and date range, accessible to ADMIN and OWNER only
- **Structured logging integration** – emit audit events to a structured logger (pino, winston) or a dedicated audit sink (e.g. a separate append-only table, an event stream) rather than the primary database, to avoid write contention on high-volume orgs

USER: Implement soft deletes for tasks instead of hard deletes. Tasks should be marked as deleted but not removed from the database. Update the relevant endpoints.

Note: Specific factual details should be independently verified. If uncertainty exists, consult authoritative sources.

Soft Deletes for Tasks

Files to Modify

```

...
src/
├── prisma/schema.prisma          # Add deletedAt to Task
├── modules/tasks/
│   └── tasks.service.ts        # Update all methods to filter/handle deletedAt
...

```

No changes to controllers, routes, validation, or audit service.

```
## `prisma/schema.prisma`
```

Add `deletedAt` to the `Task` model only. All other models are unchanged:

```
``prisma
model Task {
  id      String    @id @default(cuid())
  title   String    @db.VarChar(500)
  description String? @db.Text
  status  TaskStatus @default(TODO)
  priority TaskPriority @default(MEDIUM)
  dueDate DateTime? @map("due_date")
  deletedAt DateTime? @map("deleted_at") // null = active, non-null = soft deleted
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt

  organizationId String @map("organization_id")
  organization Organization @relation(fields: [organizationId], references: [id], onDelete: Cascade)

  creatorId String @map("creator_id")
  creator User @relation("TaskCreator", fields: [creatorId], references: [id], onDelete: Restrict)

  assigneeId String? @map("assignee_id")
  assignee User? @relation("TaskAssignee", fields: [assigneeId], references: [id], onDelete: SetNull)

  @@index([organizationId])
  @@index([organizationId, status])
  @@index([assigneeId])
  @@index([organizationId, deletedAt]) // Supports filtering active vs deleted tasks
  @@map("tasks")
}
...`
```

Run the migration:

```
```bash
npx prisma migrate dev --name add-task-soft-delete
npx prisma generate
```
```

****Why `deletedAt DateTime?` instead of `isDeleted Boolean`:**** A nullable timestamp carries more information than a boolean flag — it records *when* the deletion happened, not just *that* it happened. This is directly useful for audit purposes and for purging old soft-deleted records after a retention window. The boolean approach requires a separate `deletedAt` column anyway if you ever need the timestamp.

****Why a new index on `(organizationId, deletedAt)`:**** Every active-task query will filter `WHERE organization_id = ? AND deleted_at IS NULL`. Without this index, PostgreSQL must scan all tasks for the org and then filter out deleted ones. With it, the query can use the index to find only active tasks directly.

```
### `src/modules/tasks/tasks.service.ts`
```

Replace the full file:

```
```typescript
import prisma from "../../config/database";
import { NotFoundError, ForbiddenError } from "../../errors/HttpErrors";
import { auditService } from "../../audit/audit.service";
import { CreateTaskInput, UpdateTaskInput, ListTasksQuery } from "../../tasks.validation";
import { Prisma } from "@prisma/client";

const TASK_SELECT = {
 id: true,
 title: true,
 description: true,
 status: true,
 priority: true,
 dueDate: true,
 deletedAt: true,
 createdAt: true,
 updatedAt: true,
 organizationId: true,
 creator: {
 select: { id: true, firstName: true, lastName: true, email: true },
 },
};
```

```

 },
 assignee: {
 select: { id: true, firstName: true, lastName: true, email: true },
 },
 } as const;

// Reusable filter clause – all normal queries exclude soft-deleted tasks
const ACTIVE_TASK_FILTER = { deletedAt: null } as const;

export const tasksService = {
 async create(
 organizationId: string,
 creatorId: string,
 input: CreateTaskInput
) {
 if (input.assigneeId) {
 await assertOrgMember(input.assigneeId, organizationId);
 }

 const task = await prisma.task.create({
 data: {
 title: input.title,
 description: input.description,
 status: input.status,
 priority: input.priority,
 dueDate: input.dueDate,
 organizationId,
 creatorId,
 assigneeId: input.assigneeId ?? null,
 // deletedAt is omitted – Prisma defaults nullable fields to null
 },
 select: TASK_SELECT,
 });

 auditService
 .log({
 action: "TASK_CREATED",
 actorId: creatorId,
 organizationId,
 resourceId: task.id,
 })
 .catch((err) = {

```

```

 console.error("Audit log failed for TASK_CREATED", { taskId: task.id, actorId: creatorId, error:
err });
 });

 return task;
},

async list(organizationId: string, query: ListTasksQuery) {
 const { status, assigneeId, dueDateFrom, dueDateTo, limit, offset } = query;

 const where: Prisma.TaskWhereInput = {
 organizationId,
 ...ACTIVE_TASK_FILTER, // Exclude soft-deleted tasks from all list results
 ...(status !== undefined && { status }),
 ...(assigneeId !== undefined && { assigneeId }),
 };

 if (dueDateFrom !== undefined || dueDateTo !== undefined) {
 where.dueDate = {
 ...(dueDateFrom !== undefined && { gte: dueDateFrom }),
 ...(dueDateTo !== undefined && { lte: dueDateTo }),
 };
 }

 const [total, tasks] = await Promise.all([
 prisma.task.count({ where }),
 prisma.task.findMany({
 where,
 select: TASK_SELECT,
 orderBy: { createdAt: "desc" },
 skip: offset,
 take: limit,
 }),
]);

 const hasNextPage = offset + tasks.length > total;
 const hasPreviousPage = offset > 0;

 return {
 tasks,
 pagination: { total, limit, offset, hasNextPage, hasPreviousPage },
 };
}

```

```
},
```

```
async getById(taskId: string, organizationId: string) {
 const task = await prisma.task.findUnique({
 where: { id: taskId },
 select: { ...TASK_SELECT, organizationId: true },
 });
```

```
 // Treat soft-deleted tasks as not found – same response as a missing task
 if (!task || task.organizationId !== organizationId || task.deletedAt !== null) {
 throw new NotFoundError("Task", "TASK_NOT_FOUND");
 }
```

```
 return task;
},
```

```
async update(
 taskId: string,
 organizationId: string,
 actorId: string,
 input: UpdateTaskInput
) {
```

```
 // getById already rejects soft-deleted tasks
 const before = await tasksService.getById(taskId, organizationId);
```

```
 if (input.assignedId !== undefined && input.assignedId !== null) {
 await assertOrgMember(input.assignedId, organizationId);
 }
```

```
 const task = await prisma.task.update({
 where: { id: taskId },
 data: {
 ...(input.title !== undefined && { title: input.title }),
 ...(input.description !== undefined && { description: input.description }),
 ...(input.status !== undefined && { status: input.status }),
 ...(input.priority !== undefined && { priority: input.priority }),
 ...(input.dueDate !== undefined && { dueDate: input.dueDate }),
 ...(input.assignedId !== undefined && { assignedId: input.assignedId }),
 },
 select: TASK_SELECT,
 });
```

```

const diff = auditService.buildDiff(
 {
 title: before.title,
 description: before.description,
 status: before.status,
 priority: before.priority,
 dueDate: before.dueDate,
 assigneeld: before.assignee?.id ?? null,
 },
 {
 ...(input.title !== undefined && { title: input.title }),
 ...(input.description !== undefined && { description: input.description }),
 ...(input.status !== undefined && { status: input.status }),
 ...(input.priority !== undefined && { priority: input.priority }),
 ...(input.dueDate !== undefined && { dueDate: input.dueDate }),
 ...(input.assigneeld !== undefined && { assigneeld: input.assigneeld }),
 }
);

auditService
 .log({ action: "TASK_UPDATED", actorId, organizationId, resourceId: taskId, diff })
 .catch((err) = {
 console.error("Audit log failed for TASK_UPDATED", { taskId, actorId, error: err });
 });

return task;
},

async delete(
 taskId: string,
 organizationId: string,
 actorId: string
) {
 // getByld rejects already-deleted tasks — idempotent delete is not supported
 const task = await tasksService.getByld(taskId, organizationId);

 // Set deletedAt instead of removing the row
 await prisma.task.update({
 where: { id: taskId },
 data: { deletedAt: new Date() },
 });
}

```

```

auditService
.log({
 action: "TASK_DELETED",
 actorId,
 organizationId,
 resourceId: taskId,
 diff: {
 title: { before: task.title, after: null },
 description: { before: task.description, after: null },
 status: { before: task.status, after: null },
 priority: { before: task.priority, after: null },
 dueDate: { before: task.dueDate, after: null },
 assigneeId: { before: task.assignee?.id ?? null, after: null },
 },
})
.catch((err) = {
 console.error("Audit log failed for TASK_DELETED", { taskId, actorId, error: err });
});
},
};

```

// — Private helpers

---

```

async function assertOrgMember(userId: string, organizationId: string): Promisevoid {
 const membership = await prisma.userOrganization.findUnique({
 where: { userId_organizationId: { userId, organizationId } },
 select: { userId: true },
 });
 if (!membership) {
 throw new ForbiddenError(
 "Assignee must be a member of this organization",
 "ASSIGNEE_NOT_MEMBER"
);
 }
}
...

```

**\*\*Why `getById` treats `deletedAt !== null` as 404:\*\*** From the perspective of any caller — list, update, delete, or a direct fetch — a soft-deleted task should behave identically to a task that



## - Timestamp

**Create the schema and endpoints (create comment, list comments for a task).**

Note: Specific factual details should be independently verified. If uncertainty exists, consult authoritative sources.

# Task Comments

## Files to Create/Modify

...

```
src/
├── prisma/schema.prisma
├── modules/comments/
│ ├── comments.routes.ts # New
│ ├── comments.controller.ts # New
│ ├── comments.service.ts # New
│ └── comments.validation.ts # New
└── app.ts # Mount new routes
```

...

---

## `prisma/schema.prisma`

Add the `Comment` model. All existing models are unchanged:

```
``prisma
model Comment {
 id String @id @default(cuid())
 content String @db.Text
 createdAt DateTime @default(now())
 updatedAt DateTime @updatedAt

 taskId String @map("task_id")
 task Task @relation(fields: [taskId], references: [id], onDelete: Cascade)

 authorId String @map("author_id")
 author User @relation(fields: [authorId], references: [id], onDelete: Restrict)

 @@index([taskId])
}
```

```
 @@index([taskId, createdAt]) // Supports chronological listing per task
 @@map("comments")
}
```

Add back-relations to `Task` and `User`:

```
``prisma
model Task {
 // ... existing fields ...
 comments Comment[]
}

model User {
 // ... existing fields ...
 comments Comment[]
}
```

Run the migration:

```
``bash
npx prisma migrate dev --name add-comments
npx prisma generate
``
```

**\*\*Why `onDelete: Cascade` on `taskId`:\*\*** A comment has no meaning outside its task. When a task is soft-deleted, comments remain in the database – they are not deleted because the task row itself is not deleted. If a task is ever hard-deleted (e.g. by a purge job), cascading removes its comments automatically.

**\*\*Why `onDelete: Restrict` on `authorId`:\*\*** Consistent with `creator\_id` on tasks and `actor\_id` on audit logs – deleting a user who has authored comments would destroy historical content. `Restrict` blocks user deletion until comments are handled.

**\*\*Why no `organizationId` on `Comment`:\*\*** Organization scoping is enforced through the task. A comment belongs to a task, which belongs to an organization. Querying comments always goes through a task lookup that is already org-scoped. Adding `organizationId` directly to `Comment` would be denormalization – it would need to stay in sync with the task's org and adds a column that is never used independently.

**\*\*Why the composite index `(taskId, createdAt)`:\*\*** The primary query pattern is "all comments for this task, ordered by time." This index serves that query directly. The single-column `(taskId)` index is redundant once the composite exists – PostgreSQL can use the leftmost prefix – but is kept explicit for clarity.

---

```
`src/modules/comments/comments.validation.ts`
```

```
``typescript
```

```
import { z } from "zod";
```

```
export const createCommentSchema = z.object({
 content: z
 .string({ required_error: "Content is required" })
 .trim()
 .min(1, "Comment cannot be empty")
 .max(10000, "Comment must not exceed 10000 characters"),
});
```

```
export const listCommentsQuerySchema = z.object({
 limit: z
 .string()
 .optional()
 .transform((val) = (val !== undefined ? parseInt(val, 10) : 50))
 .pipe(
 z.number().int("Limit must be an integer").min(1).max(200, "Limit must not exceed 200")
),
});
```

```
offset: z
 .string()
 .optional()
 .transform((val) = (val !== undefined ? parseInt(val, 10) : 0))
 .pipe(
 z.number().int("Offset must be an integer").min(0, "Offset must be zero or greater")
),
});
```

```
export type CreateCommentInput = z.infer
```

```
export type ListCommentsQuery = z.infer
```

```
...
```

**\*\*Why the default `limit` is 50 for comments and not 20:\*\*** Comments on a task are typically read all at once – a user opens a task and wants to see the full discussion thread. A higher default reduces the number of requests needed to render a typical thread. The maximum is 200 rather than 100 for the same reason.

---

```
`src/modules/comments/comments.service.ts`
```

```
``typescript
```

```
import prisma from "../../config/database";
```

```
import { NotFoundError } from "../../errors/HttpErrors";
```

```
import { CreateCommentInput, ListCommentsQuery } from "./comments.validation";
```

```
const COMMENT_SELECT = {
```

```
 id: true,
```

```
 content: true,
```

```
 createdAt: true,
```

```
 updatedAt: true,
```

```
 taskId: true,
```

```
 author: {
```

```
 select: { id: true, firstName: true, lastName: true, email: true },
```

```
 },
```

```
} as const;
```

```
export const commentsService = {
```

```
 async create(
 taskId: string,
```

```
 organizationId: string,
```

```
 authorId: string,
```

```
 input: CreateCommentInput
```

```
) {
```

```
 // Verify the task exists, belongs to this org, and is not soft-deleted
```

```
 await assertActiveTask(taskId, organizationId);
```

```
 const comment = await prisma.comment.create({
```

```
 data: {
```

```
 content: input.content,
```

```
 taskId,
```

```
 authorId,
```

```
 },
```

```
 select: COMMENT_SELECT,
```

```

});

return comment;
},

async list(
 taskId: string,
 organizationId: string,
 query: ListCommentsQuery
) {
 // Verify the task exists and is accessible before listing its comments
 await assertActiveTask(taskId, organizationId);

 const { limit, offset } = query;

 const [total, comments] = await Promise.all([
 prisma.comment.count({ where: { taskId } }),
 prisma.comment.findMany({
 where: { taskId },
 select: COMMENT_SELECT,
 orderBy: { createdAt: "asc" }, // Chronological – oldest first for threaded reading
 skip: offset,
 take: limit,
 }),
]);

 const hasNextPage = offset + comments.length > total;
 const hasPreviousPage = offset > 0;

 return {
 comments,
 pagination: { total, limit, offset, hasNextPage, hasPreviousPage },
 };
};

```

// — Private helpers

---

```

async function assertActiveTask(
 taskId: string,
 organizationId: string

```

```

): Promisevoid {
 const task = await prisma.task.findUnique({
 where: { id: taskId },
 select: { organizationId: true, deletedAt: true },
 });

 if (!task || task.organizationId !== organizationId || task.deletedAt !== null) {
 throw new NotFoundError("Task", "TASK_NOT_FOUND");
 }
}
...

```

**\*\*Why `assertActiveTask` instead of calling `tasksService.getById`:\*\*** Importing `tasksService` from `commentsService` would create a cross-module dependency. `assertActiveTask` is a focused private helper that performs only the check needed here – existence, org ownership, and not soft-deleted – without pulling in the full task service and its dependencies.

**\*\*Why comments are ordered `asc` by `createdAt`:\*\*** Tasks are listed newest-first because you want to see recent work at the top. Comments are listed oldest-first because you read a conversation chronologically – the first comment provides context for everything that follows.

**\*\*Why `list` calls `assertActiveTask`:\*\*** Without this check, a member could list comments on a soft-deleted task by knowing its ID. The task is inaccessible through the task endpoints, so its comments should be inaccessible too. The check enforces consistent visibility rules.

---

```
`src/modules/comments/comments.controller.ts`
```

```

``typescript
import { Response, NextFunction } from "express";
import { createCommentSchema, listCommentsQuerySchema } from "../comments.validation";
import { commentsService } from "../comments.service";
import { OrgScopedRequest } from "../../types";

export const commentsController = {
 async create(
 req: OrgScopedRequest,
 res: Response,
 next: NextFunction
): Promisevoid {
 try {

```

```

const comment = await commentsService.create(
 req.params.taskId,
 req.membership.organizationId,
 req.user.sub,
 req.body as CreateCommentInput
);

res.status(201).json({ status: "success", data: { comment } });
} catch (err) {
 next(err);
}
},
);

async list(
 req: OrgScopedRequest,
 res: Response,
 next: NextFunction
): Promisevoid {
 try {
 const { comments, pagination } = await commentsService.list(
 req.params.taskId,
 req.membership.organizationId,
 req.query as unknown as import("./comments.validation").ListCommentsQuery
);

 res.status(200).json({ status: "success", data: { comments, pagination } });
 } catch (err) {
 next(err);
 }
},
);
...

```

---

```

`src/modules/comments/comments.routes.ts`

```

```

``typescript
import { Router } from "express";
import { Request, Response, NextFunction } from "express";
import { authenticate } from "../../middleware/authenticate";
import { requireRole } from "../../middleware/requireRole";

```

```

import { validate } from "../../middleware/validate";
import { commentsController } from "../comments.controller";
import { createCommentSchema, listCommentsQuerySchema } from "../comments.validation";
import { OrgScopedRequest } from "../../types";

const router = Router({ mergeParams: true }); // Inherits :organizationId and :taskId from parent

function asOrgScoped(
 handler: (req: OrgScopedRequest, res: Response, next: NextFunction) = Promisevoid
){
 return (req: Request, res: Response, next: NextFunction) =
 handler(req as OrgScopedRequest, res, next);
}

router.use(authenticate);

// GET /api/v1/organizations/:organizationId/tasks/:taskId/comments
// VIEWER and above – read-only access consistent with task visibility
router.get(
 "/",
 requireRole("VIEWER"),
 validate(listCommentsQuerySchema, "query"),
 asOrgScoped(commentsController.list)
);

// POST /api/v1/organizations/:organizationId/tasks/:taskId/comments
// MEMBER and above – viewers cannot comment
router.post(
 "/",
 requireRole("MEMBER"),
 validate(createCommentSchema),
 asOrgScoped(commentsController.create)
);

export default router;
...

```

**\*\*Why VIEWER cannot create comments:\*\*** A viewer has read-only access to the organization. Allowing viewers to comment would be a write operation, which is inconsistent with the VIEWER role definition established in the permission system. MEMBER is the minimum role for any write action.

---

## Updated `src/app.ts`

Add the comment routes mount below the task routes mount:

```
``typescript
import commentRoutes from "../modules/comments/comments.routes";

// Existing
app.use("/api/v1/organizations/:organizationId/tasks", taskRoutes);

// Add
app.use(
 "/api/v1/organizations/:organizationId/tasks/:taskId/comments",
 commentRoutes
);
...

```

Both `:organizationId` and `:taskId` are available inside `commentRoutes` via `mergeParams: true`.

---

## Request / Response Reference

### `POST /api/v1/organizations/:organizationId/tasks/:taskId/comments`

**\*\*Headers:\*\***

...

Authorization: Bearer token

Content-Type: application/json

...

**\*\*Request body:\*\***

``json

{

"content": "I've started working on this. Will have a draft by EOD."

}

...

**\*\*201 Created:\*\***

```
```json
{
  "status": "success",
  "data": {
    "comment": {
      "id": "clcomment123",
      "content": "I've started working on this. Will have a draft by EOD.",
      "taskId": "cltask789",
      "createdAt": "2024-01-15T14:00:00.000Z",
      "updatedAt": "2024-01-15T14:00:00.000Z",
      "author": {
        "id": "clxyz123",
        "firstName": "Jane",
        "lastName": "Doe",
        "email": "jane@example.com"
      }
    }
  }
}
```
```

**\*\*404 – task not found or soft-deleted:\*\***

```
```json
{
  "status": "error",
  "code": "TASK_NOT_FOUND",
  "message": "Task not found"
}
```
```

**\*\*422 – empty content:\*\***

```
```json
{
  "status": "error",
  "code": "VALIDATION_ERROR",
  "message": "Validation failed",
  "errors": {
    "content": ["Comment cannot be empty"]
  }
}
```
```

--

### `GET /api/v1/organizations/:organizationId/tasks/:taskId/comments`

\*\*Query params:\*\* `?limit=50&offset=0`

\*\*200 OK:\*\*

``json

```
{
 "status": "success",
 "data": {
 "comments": [
 {
 "id": "clcomment001",
 "content": "Assigned to me – picking this up now.",
 "taskId": "cltask789",
 "createdAt": "2024-01-15T09:00:00.000Z",
 "updatedAt": "2024-01-15T09:00:00.000Z",
 "author": {
 "id": "clxyz789",
 "firstName": "Bob",
 "lastName": "Smith",
 "email": "bob@example.com"
 }
 },
 {
 "id": "clcomment002",
 "content": "I've started working on this. Will have a draft by EOD.",
 "taskId": "cltask789",
 "createdAt": "2024-01-15T14:00:00.000Z",
 "updatedAt": "2024-01-15T14:00:00.000Z",
 "author": {
 "id": "clxyz123",
 "firstName": "Jane",
 "lastName": "Doe",
 "email": "jane@example.com"
 }
 }
],
 "pagination": {
 "total": 2,
 "limit": 50,

```

```
"offset": 0,
"hasNextPage": false,
"hasPreviousPage": false
}
}
}
...

```

## ## Optional / Future Enhancements

- **Edit comment** – `PATCH /comments/:commentId` restricted to the comment's own author, with the same `validate` + service pattern used for task updates
- **Delete comment** – `DELETE /comments/:commentId` where the author can delete their own comment and ADMIN can delete any comment; soft delete is optional here since comments are lower-stakes than tasks
- **Comment count on task responses** – add `\_count: { comments: true }` to `TASK\_SELECT` so task list and detail responses include a comment count without a separate request

**USER: I want task assignments to send notifications. Design a notification system where:**

- **When a task is assigned, create a notification for that user**
- **Users can retrieve their unread notifications**
- **Users can mark notifications as read**

**Show me the schema and basic endpoints.**

Note: Specific factual details should be independently verified. If uncertainty exists, consult authoritative sources.

## # Notification System

### ## Files to Create/Modify

...

```
src/
|— prisma/schema.prisma
|— modules/notifications/
| |— notifications.routes.ts # New
```

```

| |— notifications.controller.ts # New
| |— notifications.service.ts # New
| |— notifications.validation.ts # New
|— modules/tasks/
| |— tasks.service.ts # Update — trigger notification on assignment
|— app.ts # Mount new routes
...

```

```

`prisma/schema.prisma`

```

Add `NotificationType` enum and `Notification` model. All existing models are unchanged:

```

```prisma
enum NotificationType {
  TASK_ASSIGNED
}

model Notification {
  id      String      @id @default(cuid())
  type    NotificationType
  readAt  DateTime?   @map("read_at") // null = unread, non-null = read
  createdAt DateTime   @default(now())

  recipientId String @map("recipient_id")
  recipient User  @relation("ReceivedNotifications", fields: [recipientId], references: [id],
onDelete: Cascade)

  actorId String @map("actor_id")
  actor User  @relation("SentNotifications", fields: [actorId], references: [id], onDelete: Restrict)

  taskId String @map("task_id")
  task Task  @relation(fields: [taskId], references: [id], onDelete: Cascade)

  @@index([recipientId, readAt]) // Primary query: unread notifications for a user
  @@index([recipientId, createdAt]) // Secondary query: all notifications for a user, newest first
  @@map("notifications")
}
...

```

Add back-relations to `User` and `Task`:

```

``prisma
model User {
  // ... existing fields ...
  receivedNotifications Notification[] @relation("ReceivedNotifications")
  sentNotifications Notification[] @relation("SentNotifications")
}

model Task {
  // ... existing fields ...
  notifications Notification[]
}
...

```

Run the migration:

```

``bash
npx prisma migrate dev --name add-notifications
npx prisma generate
...

```

****Why `readAt DateTime` instead of `isRead Boolean`:**** Same reasoning as `deletedAt` on tasks – a timestamp carries when the event happened, not just that it happened. This enables queries like "notifications read in the last 7 days" and supports future retention/cleanup jobs.

****Why two named relations on `User`:**** A user appears twice in `Notification` – once as `recipient` and once as `actor` (the person who triggered the notification). Prisma requires named relations when two FKs point to the same model. `ReceivedNotifications` and `SentNotifications` make the direction explicit.

****Why `onDelete: Cascade` on `recipientId`:**** If a user is deleted, their notifications have no recipient and should be removed. This is safe because notifications are ephemeral UI state, not historical records.

****Why `onDelete: Restrict` on `actorId`:**** Consistent with the audit log pattern – the actor's identity is part of the notification content ("Jane assigned you to task X"). Deleting the actor would corrupt the notification's meaning.

****Why `onDelete: Cascade` on `taskId`:**** If a task is hard-deleted (e.g. by a purge job), its notifications are meaningless and should be removed. Note that soft-deleted tasks retain their row, so notifications for soft-deleted tasks remain until the task is purged.

****Why the composite index `(recipientId, readAt)`:**** The primary query is "unread notifications for this user" — `WHERE recipient_id = ? AND read_at IS NULL`. This index serves it directly. PostgreSQL can use the index to find only unread rows without scanning all notifications for the user.

```
## `src/modules/notifications/notifications.validation.ts`
```

```
``typescript
```

```
import { z } from "zod";
```

```
export const listNotificationsQuerySchema = z.object({
  unreadOnly: z
    .enum(["true", "false"], {
      errorMap: () = ({ message: "unreadOnly must be true or false" }),
    })
    .optional()
    .transform((val) = val === "true"),

  limit: z
    .string()
    .optional()
    .transform((val) = (val !== undefined ? parseInt(val, 10) : 20))
    .pipe(
      z.number().int("Limit must be an integer").min(1).max(100, "Limit must not exceed 100")
    ),

  offset: z
    .string()
    .optional()
    .transform((val) = (val !== undefined ? parseInt(val, 10) : 0))
    .pipe(
      z.number().int("Offset must be an integer").min(0, "Offset must be zero or greater")
    ),
});

export const markReadSchema = z.object({
  notificationIds: z
    .array(
      z.string().cuid("Each notification ID must be a valid ID")
    )
  )
```

```
    .min(1, "At least one notification ID is required")
    .max(100, "Cannot mark more than 100 notifications at once"),
  });
```

```
export type ListNotificationsQuery = z.inferTypeof listNotificationsQuerySchema;
export type MarkReadInput = z.inferTypeof markReadSchema;
...

```

****Why `unreadOnly` is a string enum `"true" | "false"` rather than `z.boolean()`**** Query string values are always strings. `z.boolean()` does not parse the string `true` — it only accepts actual booleans. The string enum approach accepts the query param as sent and transforms it to a boolean.

****Why `markReadSchema` accepts an array of IDs**** Marking notifications read one at a time would require one request per notification. A user opening their notification panel may have 10–20 unread items. Batch marking is more efficient and reduces round trips.

```
## `src/modules/notifications/notifications.service.ts`
```

```
``typescript
```

```
import prisma from "../../config/database";
import { ForbiddenError } from "../../errors/HttpErrors";
import { ListNotificationsQuery, MarkReadInput } from "./notifications.validation";
```

```
const NOTIFICATION_SELECT = {
  id: true,
  type: true,
  readAt: true,
  createdAt: true,
  actor: {
    select: { id: true, firstName: true, lastName: true, email: true },
  },
  task: {
    select: { id: true, title: true, organizationId: true },
  },
} as const;
```

```
export const notificationsService = {
  // Called internally by tasksService — not exposed as an HTTP endpoint
  async createTaskAssignedNotification(
```

```

    taskId: string,
    recipientId: string,
    actorId: string
  ): Promisevoid {
    // Do not notify a user who assigned a task to themselves
    if (recipientId === actorId) return;

    await prisma.notification.create({
      data: {
        type: "TASK_ASSIGNED",
        taskId,
        recipientId,
        actorId,
      },
    });
  },
};

async list(recipientId: string, query: ListNotificationsQuery) {
  const { unreadOnly, limit, offset } = query;

  const where = {
    recipientId,
    ...(unreadOnly && { readAt: null }),
  };

  const [total, notifications] = await Promise.all([
    prisma.notification.count({ where }),
    prisma.notification.findMany({
      where,
      select: NOTIFICATION_SELECT,
      orderBy: { createdAt: "desc" },
      skip: offset,
      take: limit,
    }),
  ]);

  const hasNextPage = offset + notifications.length > total;
  const hasPreviousPage = offset > 0;

  return {
    notifications,
    pagination: { total, limit, offset, hasNextPage, hasPreviousPage },
  };
}

```

```

    };
  },

  async markRead(recipientId: string, input: MarkReadInput) {
    // updateMany with recipientId in the WHERE clause ensures a user
    // can only mark their own notifications as read – not another user's
    const result = await prisma.notification.updateMany({
      where: {
        id: { in: input.notificationIds },
        recipientId,           // Ownership check
        readAt: null,         // Only update currently unread ones
      },
      data: {
        readAt: new Date(),
      },
    });

    return { markedRead: result.count };
  },
};
...

```

****Why `createTaskAssignedNotification` is `void` and not awaited by the caller:**** Notification creation is a side effect of task assignment. If the notification write fails, the task assignment itself should still succeed. The caller (task service) uses fire-and-forget with a `.catch` handler, identical to the audit log pattern.

****Why `recipientId === actorId` is checked before creating:**** If a user assigns a task to themselves, sending them a notification about their own action is noise. This guard prevents self-notifications without requiring a database round trip.

****Why `updateMany` with `recipientId` in the `WHERE` clause:**** Without the `recipientId` filter, any authenticated user could mark any notification as read by guessing or knowing its ID. The `recipientId` filter ensures the update only affects notifications that belong to the requesting user. `updateMany` returns a count rather than the updated rows, which is sufficient here – the response tells the client how many were actually marked.

****Why `readAt: null` in the `WHERE` clause of `markRead`:**** Marking an already-read notification as read again is a no-op. Including this filter means `result.count` accurately reflects how many notifications transitioned from unread to read, which is useful for the client to know.

--

```
## `src/modules/notifications/notifications.controller.ts`
```

```
``typescript
```

```
import { Request, Response, NextFunction } from "express";  
import { notificationsService } from "../notifications.service";  
import { ListNotificationsQuery, MarkReadInput } from "../notifications.validation";  
import { AuthenticatedRequest } from "../../types";
```

```
export const notificationsController = {  
  async list(  
    req: AuthenticatedRequest,  
    res: Response,  
    next: NextFunction  
  ): Promisevoid {  
    try {  
      const { notifications, pagination } = await notificationsService.list(  
        req.user.sub,  
        req.query as unknown as ListNotificationsQuery  
      );  
  
      res.status(200).json({ status: "success", data: { notifications, pagination } });  
    } catch (err) {  
      next(err);  
    }  
  },  
}
```

```
  async markRead(  
    req: AuthenticatedRequest,  
    res: Response,  
    next: NextFunction  
  ): Promisevoid {  
    try {  
      const result = await notificationsService.markRead(  
        req.user.sub,  
        req.body as MarkReadInput  
      );  
  
      res.status(200).json({ status: "success", data: result });  
    } catch (err) {  
      next(err);  
    }  
  }  
}
```

```
},  
};  
...
```

****Why these endpoints use `AuthenticatedRequest` and not `OrgScopedRequest`:****
Notifications belong to a user, not to an organization. A user's notifications span all their organizations. There is no `:organizationId` route param here – the recipient is identified solely by `req.user.sub` from the JWT.

```
## `src/modules/notifications/notifications.routes.ts`
```

```
``typescript
```

```
import { Router } from "express";  
import { Request, Response, NextFunction } from "express";  
import { authenticate } from "../../middleware/authenticate";  
import { validate } from "../../middleware/validate";  
import { notificationsController } from "./notifications.controller";  
import { listNotificationsQuerySchema, markReadSchema } from "./notifications.validation";  
import { AuthenticatedRequest } from "../../types";
```

```
const router = Router();
```

```
function asAuthenticated(  
  handler: (req: AuthenticatedRequest, res: Response, next: NextFunction) = Promisevoid  
) {  
  return (req: Request, res: Response, next: NextFunction) =  
    handler(req as AuthenticatedRequest, res, next);  
}
```

```
router.use(authenticate);
```

```
// GET /api/v1/notifications  
router.get(  
  "/",  
  validate(listNotificationsQuerySchema, "query"),  
  asAuthenticated(notificationsController.list)  
);
```

```
// PATCH /api/v1/notifications/read  
router.patch(  
  "/read",  
  validate(markReadSchema, "query"),  
  asAuthenticated(notificationsController.markRead)  
);
```

```
"/read",
validate(markReadSchema),
asAuthenticated(notificationsController.markRead)
);
```

```
export default router;
...
```

****Why `PATCH /notifications/read` and not `PATCH /notifications/:notificationId`:**** The mark-read operation is a batch action on a set of IDs provided in the request body. A single-resource URL implies one resource at a time. The `/read` sub-path makes the batch intent clear and avoids a route conflict with a hypothetical single-notification endpoint.

```
## Updated `src/modules/tasks/tasks.service.ts`
```

Add the notification call inside `update`. Only the `update` method changes – all other methods are unchanged:

```
``typescript
// Add import at the top
import { notificationsService } from "../notifications/notifications.service";

async update(
  taskId: string,
  organizationId: string,
  actorId: string,
  input: UpdateTaskInput
){
  const before = await tasksService.getById(taskId, organizationId);

  if (input.assignedId !== undefined && input.assignedId !== null) {
    await assertOrgMember(input.assignedId, organizationId);
  }

  const task = await prisma.task.update({
    where: { id: taskId },
    data: {
      ...(input.title !== undefined && { title: input.title }),
      ...(input.description !== undefined && { description: input.description }),
      ...(input.status !== undefined && { status: input.status }),
    }
  });
}
```

```

    ...(input.priority !== undefined && { priority: input.priority }),
    ...(input.dueDate !== undefined && { dueDate: input.dueDate }),
    ...(input.assigneeId !== undefined && { assigneeId: input.assigneeId }),
  },
  select: TASK_SELECT,
});

// Notify the new assignee if the assignee changed and a new one was set
const assigneeChanged =
  input.assigneeId !== undefined &&
  input.assigneeId !== null &&
  input.assigneeId !== (before.assignee?.id ?? null);

if (assigneeChanged) {
  notificationsService
    .createTaskAssignedNotification(taskId, input.assigneeId!, actorId)
    .catch((err) = {
      console.error("Notification failed for TASK_ASSIGNED", {
        taskId,
        recipientId: input.assigneeId,
        actorId,
        error: err,
      });
    });
}

// Audit log (unchanged from previous step)
const diff = auditService.buildDiff(
  {
    title: before.title,
    description: before.description,
    status: before.status,
    priority: before.priority,
    dueDate: before.dueDate,
    assigneeId: before.assignee?.id ?? null,
  },
  {
    ...(input.title !== undefined && { title: input.title }),
    ...(input.description !== undefined && { description: input.description }),
    ...(input.status !== undefined && { status: input.status }),
    ...(input.priority !== undefined && { priority: input.priority }),
    ...(input.dueDate !== undefined && { dueDate: input.dueDate }),
  }
);

```

```

    ...(input.assigneeId !== undefined && { assigneeId: input.assigneeId } ),
  }
);

auditService
  .log({ action: "TASK_UPDATED", actorId, organizationId, resourceId: taskId, diff })
  .catch((err) = {
    console.error("Audit log failed for TASK_UPDATED", { taskId, actorId, error: err });
  });

return task;
},
...

```

****Why `assigneeChanged` checks three conditions:****

| Condition | Reason |
|--|--|
| `input.assigneeId !== undefined` | The field was included in the update request |
| `input.assigneeId !== null` | The update is assigning someone, not unassigning |
| `input.assigneeId !== before.assignee?.id` | The assignee actually changed – reassigning to the same person is not a new assignment |

All three must be true to send a notification. Missing any one would produce false positives.

****Why `create` does not trigger a notification:**** Task creation with an initial `assigneeId` could also notify the assignee. This is a product decision – some teams want it, others find it noisy. It is left out of scope here and noted below as a future enhancement.

Updated `src/app.ts`

```

```typescript
import notificationRoutes from "../modules/notifications/notifications.routes";

// Add below existing routes
app.use("/api/v1/notifications", notificationRoutes);
...

```

---

## Request / Response Reference

### `GET /api/v1/notifications`

\*\*Query params:\*\* `?unreadOnly=true&limit=20&offset=0`

\*\*200 OK:\*\*

``json

```
{
 "status": "success",
 "data": {
 "notifications": [
 {
 "id": "clnotif123",
 "type": "TASK_ASSIGNED",
 "readAt": null,
 "createdAt": "2024-01-15T14:30:00.000Z",
 "actor": {
 "id": "clxyz123",
 "firstName": "Jane",
 "lastName": "Doe",
 "email": "jane@example.com"
 },
 "task": {
 "id": "cltask789",
 "title": "Write API documentation",
 "organizationId": "clorg456"
 }
 }
],
 "pagination": {
 "total": 1,
 "limit": 20,
 "offset": 0,
 "hasNextPage": false,
 "hasPreviousPage": false
 }
 }
}
```

``

--

```
`PATCH /api/v1/notifications/read`
```

```
Request body:
```

```
```json  
{  
  "notificationIds": ["clnotif123", "clnotif124"]  
}  
...`
```

```
**200 OK:**
```

```
```json  
{
 "status": "success",
 "data": {
 "markedRead": 2
 }
}
...`
```

**\*\*Why `markedRead` may be less than the number of IDs sent:\*\*** IDs that do not belong to the requesting user, or that are already read, are silently excluded by the `WHERE` clause. The count reflects only the notifications that actually transitioned from unread to read.

--

```
Data Flow: Task Assignment -> Notification
```

```
...
```

```
PATCH /api/v1/organizations/:organizationId/tasks/:taskId
```

|



```
tasksService.update(taskId, organizationId, actorId, input)
```

|

— fetch task before state

— prisma.task.update(...)

— assigneeChanged?

  |— no → skip

  |— yes → notificationsService

    .createTaskAssignedNotification(taskId, newAssigneeId, actorId)

    .catch(console.error) ← fire-and-forget

```
... └─ auditService.log(...).catch(console.error) ← fire-and-forget
...

```

## ## Optional / Future Enhancements

- **Notify on task creation with initial assignee** – call `createTaskAssignedNotification`` inside `tasksService.create`` when `input.assigneeId`` is set
- **Unread count endpoint** – `GET /api/v1/notifications/unread-count`` returning `{ count: number }` for badge display in a UI, using `prisma.notification.count({ where: { recipientId, readAt: null } })``
- **Mark all read** – `PATCH /api/v1/notifications/read-all`` that calls `prisma.notification.updateMany({ where: { recipientId, readAt: null }, data: { readAt: new Date() } })`` without requiring an ID list
- **Additional notification types** – `TASK_COMMENTED``, `TASK_STATUS_CHANGED``, `TASK_DUE_SOON`` – each would add a value to the `NotificationType`` enum and a corresponding trigger in the relevant service method
- **Notification preferences** – a `NotificationPreference`` model per user per type, allowing users to opt out of specific notification types before the `createTaskAssignedNotification`` call writes to the database
- **Real-time delivery** – the current implementation is pull-based (client polls `GET /notifications``). A WebSocket or Server-Sent Events layer could push notifications to connected clients immediately after `prisma.notification.create`` resolves
- **Retention / cleanup job** – a scheduled job that hard-deletes notifications older than N days where `readAt IS NOT NULL``, keeping the table small over time

**USER:Add rate limiting to prevent API abuse. Registration and login endpoints should be more strictly limited than other endpoints.**

Note: Specific factual details should be independently verified. If uncertainty exists, consult authoritative sources.

# Rate Limiting

## Dependencies

```bash

```
npm install express-rate-limit
npm install --save-dev @types/express-rate-limit
...
```

`express-rate-limit` is the standard rate limiting library for Express. It stores counters in memory by default, which is sufficient for a single-process deployment.

```
## Files to Create/Modify
```

...

```
src/
├── config/
│   └── rateLimiter.ts      # New — rate limiter instances
├── middleware/
│   └── errorHandler.ts    # Update — handle rate limit errors consistently
├── modules/auth/
│   └── auth.routes.ts     # Apply strict limiter
└── app.ts                 # Apply default limiter globally
...
```

```
## `src/config/rateLimiter.ts`
```

```
``typescript
import rateLimit, { RateLimitRequestHandler } from "express-rate-limit";

// Default limiter — applied globally to all routes
export const defaultLimiter: RateLimitRequestHandler = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 200, // 200 requests per window per IP
  standardHeaders: "draft-7", // Emit RateLimit headers per IETF draft-7
  legacyHeaders: false, // Suppress X-RateLimit-* headers (older format)
  message: {
    status: "error",
    code: "RATE_LIMIT_EXCEEDED",
    message: "Too many requests. Please try again later.",
  },
  skipSuccessfulRequests: false,
});
```

```

// Auth limiter – applied to /auth/register and /auth/login only
export const authLimiter: RateLimitRequestHandler = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 10, // 10 attempts per window per IP
  standardHeaders: "draft-7",
  legacyHeaders: false,
  message: {
    status: "error",
    code: "RATE_LIMIT_EXCEEDED",
    message: "Too many authentication attempts. Please try again in 15 minutes.",
  },
  skipSuccessfulRequests: false,
});
...

```

****Why `standardHeaders: "draft-7"` and `legacyHeaders: false`:**** The IETF `RateLimit` header draft (draft-7 at time of writing – verify current draft status independently) standardises how rate limit information is communicated to clients. Emitting the standard headers allows clients and API gateways to read limit state without custom parsing. The legacy `X-RateLimit-*` headers are suppressed to avoid sending duplicate information in two formats.

****Why `skipSuccessfulRequests: false`:**** Successful requests count toward the limit. Setting this to `true` would allow an attacker to make unlimited successful login attempts – only failed ones would be counted. For auth endpoints especially, every attempt must count.

****Why 10 attempts per 15 minutes for auth:**** This is a commonly cited starting point for login rate limiting. At 10 attempts per 15 minutes, a brute-force attack against an 8-character alphanumeric password is computationally infeasible within the window. The exact numbers should be tuned based on observed legitimate user behaviour – if users frequently hit the limit, the window or count needs adjustment.

****Why 200 requests per 15 minutes for the default:**** This accommodates normal API usage patterns for a task management application. A user actively working – loading task lists, updating statuses, adding comments – would typically make far fewer than 200 requests in 15 minutes.

```
## Updated `src/modules/auth/auth.routes.ts`
```

```

```typescript
import { Router } from "express";

```

```

import { authController } from "./auth.controller";
import { validate } from "../middleware/validate";
import { registerSchema, loginSchema } from "./auth.validation";
import { authLimiter } from "../config/rateLimiter";

const router = Router();

router.post(
 "/register",
 authLimiter,
 validate(registerSchema),
 authController.register
);

router.post(
 "/login",
 authLimiter,
 validate(loginSchema),
 authController.login
);

export default router;

```

**\*\*Why `authLimiter` is applied per-route and not at the router level:\*\*** The auth router may gain public routes in the future (e.g. a password reset request endpoint) that warrant their own limit. Per-route application keeps the intent explicit.

**\*\*Why `authLimiter` is placed before `validate`:\*\*** A rate-limited request should be rejected before any processing occurs – including validation and database queries. Placing the limiter first minimises work done for requests that will be rejected anyway.

---

## Updated `src/app.ts`

```

```typescript
import express from "express";
import { env } from "./config/env";
import { errorHandler } from "./middleware/errorHandler";
import { defaultLimiter } from "./config/rateLimiter";
import authRoutes from "./modules/auth/auth.routes";

```

```

import organizationRoutes from "./modules/organizations/organizations.routes";
import taskRoutes from "./modules/tasks/tasks.routes";
import commentRoutes from "./modules/comments/comments.routes";
import notificationRoutes from "./modules/notifications/notifications.routes";

const app = express();

app.use(express.json());

// Apply default rate limiter to all routes
// Auth routes apply their own stricter limiter on top of this
app.use(defaultLimiter);

app.get("/health", (req, res) = {
  res.json({ status: "ok", environment: env.NODE_ENV });
});

app.use("/api/v1/auth", authRoutes);
app.use("/api/v1/organizations", organizationRoutes);
app.use("/api/v1/organizations/:organizationId/tasks", taskRoutes);
app.use("/api/v1/organizations/:organizationId/tasks/:taskId/comments", commentRoutes);
app.use("/api/v1/notifications", notificationRoutes);

app.use(errorHandler);

app.listen(env.PORT, () = {
  console.log(`🚀 Server running on port ${env.PORT} [${env.NODE_ENV}]`);
});

export default app;

```

****Why `defaultLimiter` is applied with `app.use` before route mounts:**** Applying it globally before any route means every endpoint – including ones added in the future – is covered without requiring each route file to remember to add it. The auth routes then apply `authLimiter` as an additional, stricter layer on top.

****Why the health check endpoint is also rate-limited:**** The health check is mounted after `app.use(defaultLimiter)`, so it is covered. This is intentional – an attacker hammering the health endpoint should be limited the same as any other endpoint. If a load balancer needs unrestricted access to `/health`, it should bypass the API's rate limiter at the infrastructure level rather than exempting the route in application code.

Rate Limit Response Headers

When a client is within the limit, every response includes:

...

```
RateLimit-Limit: 200
RateLimit-Remaining: 187
RateLimit-Reset: 1705320000
...
```

When the limit is exceeded, the response is:

...

```
HTTP/1.1 429 Too Many Requests
RateLimit-Limit: 10
RateLimit-Remaining: 0
RateLimit-Reset: 1705320000
Retry-After: 847
```

```
{
  "status": "error",
  "code": "RATE_LIMIT_EXCEEDED",
  "message": "Too many authentication attempts. Please try again in 15 minutes."
}
...
```

****Why the error body matches the existing error response shape:**** The `message` object passed to `express-rate-limit` is sent directly as the response body when the limit is exceeded. Matching the `{ status, code, message }` shape means clients can handle 429 responses with the same error-parsing logic they use for 400, 401, 403, and 404 responses.

Effective Limits Per Endpoint

...

| Endpoint | Window | Max requests |
|----------|--------|--------------|
|----------|--------|--------------|

```
POST /api/v1/auth/register      15 min   10 (authLimiter)
POST /api/v1/auth/login         15 min   10 (authLimiter)
All other endpoints            15 min   200 (defaultLimiter)
...

```

Auth endpoints receive both limiters — `defaultLimiter` from `app.use` and `authLimiter` from the route. The stricter limit (10) is the effective ceiling because `authLimiter` will reject the request before `defaultLimiter`'s counter reaches 200.

Optional / Future Enhancements

- **Redis-backed store** — the default in-memory store does not share state across multiple Node.js processes or instances. In a multi-instance deployment, use `rate-limit-redis` with an `ioredis` client so all instances share the same counter. This requires adding Redis to the infrastructure
- **Per-user limiting** — the default key is the client IP address. Authenticated endpoints could use `req.user.sub` as the key so limits apply per user account rather than per IP, which is more accurate for users behind shared NAT or proxies
- **Dynamic limits** — different limits for different subscription tiers (e.g. free vs paid org members), implemented by reading the org's plan from the database in a custom `keyGenerator` function

USER: Write unit tests for the authentication middleware we created in prompt 4. Use Jest and show me at least 3 test cases covering different scenarios.

Note: Specific factual details should be independently verified. If uncertainty exists, consult authoritative sources.

Unit Tests for Authentication Middleware

Dependencies

```
```bash
npm install --save-dev jest ts-jest @types/jest
...

```

---

```
`jest.config.ts`
```

```
``typescript
```

```
import type { Config } from "jest";
```

```
const config: Config = {
 preset: "ts-jest",
 testEnvironment: "node",
 roots: [<rootDir>/src],
 testMatch: ["**/_tests_/**/*test.ts"],
 clearMocks: true, // Reset mock state between tests automatically
};
```

```
export default config;
```

```
``
```

Add to `package.json` scripts:

```
``json
```

```
"test": "jest",
```

```
"test:watch": "jest --watch"
```

```
``
```

```

```

```
`src/middleware/_tests_/authenticate.test.ts`
```

```
``typescript
```

```
import { Request, Response, NextFunction } from "express";
```

```
import jwt from "jsonwebtoken";
```

```
import { authenticate } from "../authenticate";
```

```
import { UnauthorizedError } from "../errors/HttpErrors";
```

```
// Mock the env module so tests are not dependent on a real .env file
```

```
jest.mock("../config/env", () => ({
```

```
 env: {
```

```
 JWT_SECRET: "test-secret-minimum-32-characters-long",
```

```
 JWT_EXPIRES_IN: "7d",
```

```
 NODE_ENV: "test",
```

```
 PORT: "3000",
```

```
 DATABASE_URL: "postgresql://test",
```

```
},
});
```

```
// — Test helpers
```

---

```
function makeReq(authHeader?: string): PartialRequest {
 return {
 headers: authHeader ? { authorization: authHeader } : {},
 };
}
```

```
function makeRes(): PartialResponse {
 return {};
}
```

```
function makeNext(): jest.MockedFunctionNextFunction {
 return jest.fn();
}
```

```
function signToken(
 payload: object,
 secret = "test-secret-minimum-32-characters-long",
 options: jwt.SignOptions = { expiresIn: "1h" }
): string {
 return jwt.sign(payload, secret, options);
}
```

```
// — Tests
```

---

```
describe("authenticate middleware", () = {
```

```
 // — Missing / malformed header
```

---

```
 describe("when Authorization header is absent", () = {
 it("calls next with UnauthorizedError", () = {
 const req = makeReq(); // No Authorization header
 const res = makeRes();
 const next = makeNext();
```

```

authenticate(req as Request, res as Response, next);

expect(next).toHaveBeenCalledTimes(1);

const err = next.mock.calls[0][0];
expect(err).toBeInstanceOf(UnauthorizedError);
expect(err.statusCode).toBe(401);
expect(err.code).toBe("MISSING_TOKEN");
});
});

describe("when Authorization header is present but not Bearer format", () = {
 it("calls next with UnauthorizedError", () = {
 const req = makeReq("Basic dXNlcjpwYXNz");
 const res = makeRes();
 const next = makeNext();

 authenticate(req as Request, res as Response, next);

 expect(next).toHaveBeenCalledTimes(1);

 const err = next.mock.calls[0][0];
 expect(err).toBeInstanceOf(UnauthorizedError);
 expect(err.statusCode).toBe(401);
 expect(err.code).toBe("MISSING_TOKEN");
 });
});

// — Invalid token

```

---

```

describe("when token is signed with the wrong secret", () = {
 it("calls next with UnauthorizedError and TOKEN_INVALID code", () = {
 const token = signToken({ sub: "user_123", email: "a@b.com" },
"wrong-secret-that-is-long-enough");
 const req = makeReq(`Bearer ${token}`);
 const res = makeRes();
 const next = makeNext();

 authenticate(req as Request, res as Response, next);

```

```

expect(next).toHaveBeenCalledTimes(1);

const err = next.mock.calls[0][0];
expect(err).toBeInstanceOf(UnauthorizedError);
expect(err.statusCode).toBe(401);
expect(err.code).toBe("TOKEN_INVALID");
});
});

describe("when token is structurally malformed", () = {
 it("calls next with UnauthorizedError and TOKEN_INVALID code", () = {
 const req = makeReq("Bearer this.is.notavalidjwt");
 const res = makeRes();
 const next = makeNext();

 authenticate(req as Request, res as Response, next);

 expect(next).toHaveBeenCalledTimes(1);

 const err = next.mock.calls[0][0];
 expect(err).toBeInstanceOf(UnauthorizedError);
 expect(err.statusCode).toBe(401);
 expect(err.code).toBe("TOKEN_INVALID");
 });
});

// — Expired token

```

---

```

describe("when token is expired", () = {
 it("calls next with UnauthorizedError and TOKEN_EXPIRED code", () = {
 // expiresIn: 0 produces a token that is expired immediately
 const token = signToken(
 { sub: "user_123", email: "a@b.com" },
 "test-secret-minimum-32-characters-long",
 { expiresIn: 0 }
);

 const req = makeReq(`Bearer ${token}`);
 const res = makeRes();
 const next = makeNext();

```

```

authenticate(req as Request, res as Response, next);

expect(next).toHaveBeenCalledTimes(1);

const err = next.mock.calls[0][0];
expect(err).toBeInstanceOf(UnauthorizedError);
expect(err.statusCode).toBe(401);
expect(err.code).toBe("TOKEN_EXPIRED");
});
});

// — Valid token

```

---

```

describe("when token is valid", () = {
 it("attaches decoded payload to req.user and calls next with no arguments", () => {
 const payload = { sub: "user_123", email: "jane@example.com" };
 const token = signToken(payload);
 const req = makeReq(`Bearer ${token}`) as Request & { user?: unknown };
 const res = makeRes();
 const next = makeNext();

 authenticate(req as Request, res as Response, next);

 expect(next).toHaveBeenCalledTimes(1);
 expect(next).toHaveBeenCalledWith(); // Called with no arguments – no error

 expect(req.user).toBeDefined();
 expect((req.user as { sub: string }).sub).toBe("user_123");
 expect((req.user as { email: string }).email).toBe("jane@example.com");
 });

 it("does not call next with an error when token is valid", () = {
 const token = signToken({ sub: "user_456", email: "bob@example.com" });
 const req = makeReq(`Bearer ${token}`);
 const res = makeRes();
 const next = makeNext();

 authenticate(req as Request, res as Response, next);

 const callArg = next.mock.calls[0][0];
 expect(callArg).toBeUndefined();
 });
});

```

```
});
});
```

```
});
...

```

## ## Test Case Summary

```
| Test | Scenario | Expected behaviour |
|--|--|--|
| No `Authorization` header | Header entirely absent | `next(UnauthorizedError)` with code `MISSING_TOKEN` |
| Non-Bearer scheme | `Basic ...` instead of `Bearer ...` | `next(UnauthorizedError)` with code `MISSING_TOKEN` |
| Wrong signing secret | Token signed with a different secret | `next(UnauthorizedError)` with code `TOKEN_INVALID` |
| Malformed token | Not a valid JWT structure | `next(UnauthorizedError)` with code `TOKEN_INVALID` |
| Expired token | `expiresIn: 0` produces immediate expiry | `next(UnauthorizedError)` with code `TOKEN_EXPIRED` |
| Valid token | Correct secret, not expired | `req.user` populated, `next()` called with no arguments |
| Valid token – no error | Same as above, explicit assertion | `next.mock.calls[0][0]` is `undefined` |

```

## ## Why These Test Cases

**\*\*Missing header and non-Bearer format\*\*** are tested separately because they hit different branches of the header check:

```
```typescript
if (!authHeader || !authHeader.startsWith("Bearer ")) { ... }
...

```

Both produce the same error, but testing both confirms neither branch is accidentally skipped.

****Wrong secret vs malformed token**** both produce ``JsonWebTokenError`` from the ``jsonwebtoken`` library, but via different internal paths. Testing both confirms the catch block handles the full range of ``JsonWebTokenError`` cases.

****Expired token**** produces ``TokenExpiredError``, which is a subclass of ``JsonWebTokenError``. Testing it separately confirms the ``instanceof jwt.TokenExpiredError`` check runs before the generic ``JsonWebTokenError`` check – if the order were reversed, expired tokens would return ``TOKEN_INVALID`` instead of ``TOKEN_EXPIRED``.

****Valid token – two assertions**** are written as separate tests rather than one test with two ``expect`` calls. If ``req.user`` is not populated, the first test fails with a clear message. If ``next`` is called with an error, the second test fails with a clear message. A single combined test would fail at the first failing assertion and hide the second.

Running the Tests

```
``bash
npm test
```

Output:

```
# PASS src/middleware/__tests__/authenticate.test.ts
# authenticate middleware
# when Authorization header is absent
#   ✓ calls next with UnauthorizedError (4ms)
# when Authorization header is present but not Bearer format
#   ✓ calls next with UnauthorizedError (1ms)
# when token is signed with the wrong secret
#   ✓ calls next with UnauthorizedError and TOKEN_INVALID code (2ms)
# when token is structurally malformed
#   ✓ calls next with UnauthorizedError and TOKEN_INVALID code (1ms)
# when token is expired
#   ✓ calls next with UnauthorizedError and TOKEN_EXPIRED code (1ms)
# when token is valid
#   ✓ attaches decoded payload to req.user and calls next with no arguments (1ms)
#   ✓ does not call next with an error when token is valid (1ms)
...

```

Optional / Future Enhancements

- **Tests for `requireRole` middleware** – mock `prisma.userOrganization.findUnique` using `jest.mock("../config/database")` and test the not-a-member (404), insufficient-role (403), and valid-member paths
- **Tests for `authService.login`** – mock `prisma.user.findUnique` and `bcrypt.compare` to test the timing-safe path (user not found still runs bcrypt), invalid credentials (401), and successful login (token returned)

USER: Write integration tests for the task creation endpoint. Test:

- **Successful task creation**
- **Unauthorized access (no token)**
- **Forbidden access (user not in organization)**
- **Validation errors (missing required fields)**

Note: Specific factual details should be independently verified. If uncertainty exists, consult authoritative sources.

Integration Tests for Task Creation

Dependencies

```
```bash
npm install --save-dev supertest @types/supertest
```
```

`supertest` makes HTTP requests against the Express app without starting a real server. `jest` and `ts-jest` are already installed from the previous step.

Files to Create

...

```
src/
├── modules/tasks/
│   └── __tests__/
│       └── tasks.create.integration.test.ts
└──
```

Test Strategy

Integration tests for this endpoint exercise the full middleware chain – `authenticate`, `requireRole`, `validate`, controller, service – against a real test database. The alternative (mocking Prisma) would test less and require maintaining mock implementations that drift from real behaviour.

Prerequisites:

- A separate test database configured via `DATABASE_URL` in a `.env.test` file
- Prisma migrations applied to the test database

```
```bash
.env.test
DATABASE_URL="postgresql://user:password@localhost:5432/taskdb_test"
```
```

Update `jest.config.ts` to load the test environment:

```
```typescript
import type { Config } from "jest";

const config: Config = {
 preset: "ts-jest",
 testEnvironment: "node",
 roots: ["<rootDir>/src"],
 testMatch: ["**/_tests_/**/*test.ts"],
 clearMocks: true,
 setupFiles: ["<rootDir>/src/test/loadEnv.ts"],
};
```

```
export default config;
```

```
```
```

```
---
```

`src/test/loadEnv.ts`

```
```typescript
import { config } from "dotenv";
import path from "path";
```

```
// Load .env.test before any module imports resolve env vars
```

```
config({ path: path.resolve(process.cwd(), ".env.test")});
`
```

```

```

```
`src/test/helpers.ts`
```

Shared setup helpers used across integration test files:

```
``typescript
import prisma from "../config/database";
import bcrypt from "bcryptjs";
import jwt from "jsonwebtoken";
import { env } from "../config/env";

export interface TestUser {
 id: string;
 email: string;
 token: string;
}

export interface TestOrg {
 id: string;
 slug: string;
}

export async function createTestUser(
 email = `test-${Date.now()}@example.com`
): PromiseTestUser {
 const user = await prisma.user.create({
 data: {
 email,
 passwordHash: await bcrypt.hash("TestPass1", 10),
 firstName: "Test",
 lastName: "User",
 },
 select: { id: true, email: true },
 });

 const token = jwt.sign(
 { sub: user.id, email: user.email },
 env.JWT_SECRET,
```

```

 { expiresIn: "1h" }
);

 return { id: user.id, email: user.email, token };
}

export async function createTestOrg(
 ownerId: string,
 slug = `test-org-${Date.now()}`
): PromiseTestOrg {
 const org = await prisma.organization.create({
 data: {
 name: "Test Organization",
 slug,
 memberships: {
 create: { userId: ownerId, role: "OWNER" },
 },
 },
 select: { id: true, slug: true },
 });

 return { id: org.id, slug: org.slug };
}

export async function addOrgMember(
 userId: string,
 organizationId: string,
 role: "ADMIN" | "MEMBER" | "VIEWER" = "MEMBER"
): Promisevoid {
 await prisma.userOrganization.create({
 data: { userId, organizationId, role },
 });
}

// Wipe all test data in dependency order after each test
export async function cleanDatabase(): Promisevoid {
 await prisma.notification.deleteMany();
 await prisma.auditLog.deleteMany();
 await prisma.comment.deleteMany();
 await prisma.task.deleteMany();
 await prisma.userOrganization.deleteMany();
 await prisma.organization.deleteMany();
}

```

```
 await prisma.user.deleteMany();
 }
 ...
```

**\*\*Why `Date.now()` in default email and slug values:\*\*** Each test run creates real database rows. Using a timestamp suffix ensures no two test runs collide on unique constraints, even if `cleanDatabase` is not called between runs during development.

**\*\*Why `bcrypt` cost factor 10 in tests:\*\*** The production value is 12. In tests, bcrypt is called during user creation setup – not under test. Using 10 reduces setup time without affecting what is being tested.

---

```
`src/modules/tasks/__tests__/tasks.create.integration.test.ts`
```

```
``typescript
```

```
import request from "supertest";
import app from "../../app";
import prisma from "../../config/database";
import {
 createTestUser,
 createTestOrg,
 addOrgMember,
 cleanDatabase,
 TestUser,
 TestOrg,
} from "../../test/helpers";
```

```
describe("POST /api/v1/organizations/:organizationId/tasks", () = {
 let owner: TestUser;
 let member: TestUser;
 let viewer: TestUser;
 let outsider: TestUser;
 let org: TestOrg;
```

```
 beforeEach(async () = {
 await cleanDatabase();
```

```
 owner = await createTestUser("owner@example.com");
 member = await createTestUser("member@example.com");
 viewer = await createTestUser("viewer@example.com");
```

```

outsider = await createTestUser("outsider@example.com");

org = await createTestOrg(owner.id);

await addOrgMember(member.id, org.id, "MEMBER");
await addOrgMember(viewer.id, org.id, "VIEWER");
// outsider has no membership in org
});

afterAll(async () = {
 await cleanDatabase();
 await prisma.$disconnect();
});

// — Successful creation

```

---

```

describe("when the request is valid", () = {
 it("returns 201 with the created task when called by an OWNER", async () = {
 const response = await request(app)
 .post(`/api/v1/organizations/${org.id}/tasks`)
 .set("Authorization", `Bearer ${owner.token}`)
 .send({
 title: "Write documentation",
 description: "Cover all endpoints",
 status: "TODO",
 priority: "HIGH",
 });

 expect(response.status).toBe(201);
 expect(response.body.status).toBe("success");

 const task = response.body.data.task;
 expect(task.id).toBeDefined();
 expect(task.title).toBe("Write documentation");
 expect(task.description).toBe("Cover all endpoints");
 expect(task.status).toBe("TODO");
 expect(task.priority).toBe("HIGH");
 expect(task.organizationId).toBe(org.id);
 expect(task.creator.id).toBe(owner.id);
 expect(task.assignee).toBeNull();
 });
});

```

```
it("returns 201 with the created task when called by a MEMBER", async () = {
 const response = await request(app)
 .post(`/api/v1/organizations/${org.id}/tasks`)
 .set("Authorization", `Bearer ${member.token}`)
 .send({ title: "Member task" });

 expect(response.status).toBe(201);
 expect(response.body.data.task.creator.id).toBe(member.id);
});
```

```
it("persists the task to the database", async () = {
 await request(app)
 .post(`/api/v1/organizations/${org.id}/tasks`)
 .set("Authorization", `Bearer ${owner.token}`)
 .send({ title: "Persisted task" });
```

```
 const task = await prisma.task.findFirst({
 where: { title: "Persisted task", organizationId: org.id },
 });
```

```
 expect(task).not.toBeNull();
 expect(task!.deletedAt).toBeNull();
});
```

```
it("applies default status TODO and priority MEDIUM when not provided", async () = {
 const response = await request(app)
 .post(`/api/v1/organizations/${org.id}/tasks`)
 .set("Authorization", `Bearer ${owner.token}`)
 .send({ title: "Defaults test" });
```

```
 expect(response.status).toBe(201);
 expect(response.body.data.task.status).toBe("TODO");
 expect(response.body.data.task.priority).toBe("MEDIUM");
});
```

```
it("accepts a valid assigneeId when the assignee is an org member", async () = {
 const response = await request(app)
 .post(`/api/v1/organizations/${org.id}/tasks`)
 .set("Authorization", `Bearer ${owner.token}`)
 .send({
 title: "Assigned task",
```

```
 assigneeId: member.id,
 });

 expect(response.status).toBe(201);
 expect(response.body.data.task.assignee.id).toBe(member.id);
});
```

```
// — Unauthorized — no token
```

---

```
describe("when no Authorization header is provided", () = {
 it("returns 401", async () = {
 const response = await request(app)
 .post(`/api/v1/organizations/${org.id}/tasks`)
 .send({ title: "Should fail" });

 expect(response.status).toBe(401);
 expect(response.body.status).toBe("error");
 expect(response.body.code).toBe("MISSING_TOKEN");
 });
```

```
 it("does not create a task in the database", async () = {
 await request(app)
 .post(`/api/v1/organizations/${org.id}/tasks`)
 .send({ title: "Should not persist" });

 const count = await prisma.task.count({
 where: { title: "Should not persist" },
 });

 expect(count).toBe(0);
 });
});
```

```
// — Forbidden — user not in organization
```

---

```
describe("when the user is not a member of the organization", () = {
 it("returns 404 for a user with no membership", async () = {
 const response = await request(app)
 .post(`/api/v1/organizations/${org.id}/tasks`)
```

```

 .set("Authorization", `Bearer ${outsider.token}`)
 .send({ title: "Outsider task" });

// 404 not 403 — do not reveal the org exists to non-members
expect(response.status).toBe(404);
expect(response.body.status).toBe("error");
expect(response.body.code).toBe("ORG_NOT_FOUND");
});

it("returns 403 for a VIEWER (read-only role)", async () = {
 const response = await request(app)
 .post(`/api/v1/organizations/${org.id}/tasks`)
 .set("Authorization", `Bearer ${viewer.token}`)
 .send({ title: "Viewer task" });

 expect(response.status).toBe(403);
 expect(response.body.status).toBe("error");
 expect(response.body.code).toBe("INSUFFICIENT_ROLE");
});

it("does not create a task when the user is an outsider", async () = {
 await request(app)
 .post(`/api/v1/organizations/${org.id}/tasks`)
 .set("Authorization", `Bearer ${outsider.token}`)
 .send({ title: "Outsider task" });

 const count = await prisma.task.count({
 where: { title: "Outsider task", organizationId: org.id },
 });

 expect(count).toBe(0);
});

// — Validation errors

describe("when the request body fails validation", () = {
 it("returns 422 when title is missing", async () = {
 const response = await request(app)
 .post(`/api/v1/organizations/${org.id}/tasks`)
 .set("Authorization", `Bearer ${owner.token}`)

```

---

```
.send({ description: "No title provided" });

expect(response.status).toBe(422);
expect(response.body.status).toBe("error");
expect(response.body.code).toBe("VALIDATION_ERROR");
expect(response.body.errors.title).toBeDefined();
});

it("returns 422 when title is an empty string", async () = {
 const response = await request(app)
 .post(`/api/v1/organizations/${org.id}/tasks`)
 .set("Authorization", `Bearer ${owner.token}`)
 .send({ title: "" });

 expect(response.status).toBe(422);
 expect(response.body.errors.title).toBeDefined();
});

it("returns 422 when status is not a valid enum value", async () = {
 const response = await request(app)
 .post(`/api/v1/organizations/${org.id}/tasks`)
 .set("Authorization", `Bearer ${owner.token}`)
 .send({ title: "Valid title", status: "INVALID_STATUS" });

 expect(response.status).toBe(422);
 expect(response.body.errors.status).toBeDefined();
});

it("returns 422 when priority is not a valid enum value", async () = {
 const response = await request(app)
 .post(`/api/v1/organizations/${org.id}/tasks`)
 .set("Authorization", `Bearer ${owner.token}`)
 .send({ title: "Valid title", priority: "URGENT" });

 expect(response.status).toBe(422);
 expect(response.body.errors.priority).toBeDefined();
});

it("returns 422 when title exceeds 500 characters", async () = {
 const response = await request(app)
 .post(`/api/v1/organizations/${org.id}/tasks`)
 .set("Authorization", `Bearer ${owner.token}`)
```

```

 .send({ title: "a".repeat(501) });

 expect(response.status).toBe(422);
 expect(response.body.errors.title).toBeDefined();
 });

 it("returns 403 when assigneeId belongs to a user outside the org", async () = {
 const response = await request(app)
 .post(`/api/v1/organizations/${org.id}/tasks`)
 .set("Authorization", `Bearer ${owner.token}`)
 .send({
 title: "Task with outsider assignee",
 assigneeId: outsider.id,
 });

 expect(response.status).toBe(403);
 expect(response.body.code).toBe("ASSIGNEE_NOT_MEMBER");
 });
});
...

```

## ## Test Coverage Map

| Test                          | Layer exercised      | What it verifies                         |
|-------------------------------|----------------------|------------------------------------------|
| OWNER creates task            | Full stack           | Happy path, response shape, `creator.id` |
| MEMBER creates task           | Full stack           | MEMBER role is sufficient for create     |
| Task persists to DB           | Full stack + DB      | Side effect — row actually written       |
| Defaults applied              | Validation + service | `status` and `priority` defaults         |
| Valid assigneeId              | Service + DB         | Assignee relation resolved and returned  |
| No token   `authenticate`     | 401                  | before any org or task logic runs        |
| No token — no DB write        | `authenticate` + DB  | Side effect blocked                      |
| Outsider user   `requireRole` | 404                  | — org existence not revealed             |
| VIEWER role   `requireRole`   | 403                  | — role hierarchy enforced                |
| Outsider — no DB write        | `requireRole` + DB   | Side effect blocked                      |
| Missing title   `validate`    | 422                  | with field-level error                   |
| Empty title   `validate`      | 422                  | — `.min(1)` enforced                     |
| Invalid status   `validate`   | 422                  | — enum validation                        |
| Invalid priority   `validate` | 422                  | — enum validation                        |

| Title too long | `validate` | 422 – `.max(500)` enforced |  
| Assignee not in org | `tasksService` | 403 – cross-org assignment blocked |

---

## ## Running the Tests

```
```bash
# Run all tests
npm test

# Run only this file
npm test -- tasks.create.integration.test

# Run with coverage
npm test -- --coverage
```
```

---

## ## Why `beforeEach` Cleans and Recreates

Each test gets a fresh database state. This prevents test ordering dependencies – a task created in test 3 cannot affect test 7. The cost is setup time per test. For integration tests against a local database, this is acceptable. If test suite time becomes a concern, `beforeAll` with targeted cleanup per test is an alternative, but it requires more careful isolation.

**Why `afterAll` disconnects Prisma:** Jest does not automatically close the Prisma connection pool. Without `prisma.\$disconnect()`, Jest may hang after tests complete waiting for the pool to drain. `afterAll` runs once after all tests in the file finish.

## ## Optional / Future Enhancements

- **Test for audit log creation** – after a successful task creation, query `prisma.auditLog.findFirst({ where: { resourceId: task.id } })` and assert the `TASK\_CREATED` entry was written
- **Test for notification on assignment** – after creating a task with an `assigneeId`, assert a `Notification` row exists for the assignee
- **Shared `app` instance** – if multiple integration test files import `app`, each import starts a new listener. Extract the `app.listen` call from `app.ts` into a separate `server.ts` entry point so

`app.ts` exports the Express app without binding to a port, which is the standard pattern for `supertest` usage

**Add API documentation. Choose a documentation approach (OpenAPI/Swagger, JSDoc, or other) and document the task endpoints we've built. Keep it consistent with our project structure.**

Note: Specific factual details should be independently verified. If uncertainty exists, consult authoritative sources.

# API Documentation with OpenAPI / Swagger

OpenAPI is the correct choice for this project. It produces machine-readable documentation that can be rendered as interactive UI, used to generate client SDKs, and validated against actual responses. JSDoc comments on route handlers would be less discoverable and harder to keep in sync.

---

## Dependencies

```
```bash
npm install swagger-ui-express yaml
npm install --save-dev @types/swagger-ui-express
```
```

`swagger-ui-express` serves the interactive Swagger UI. `yaml` parses the OpenAPI spec file. The spec itself is written in YAML – it is easier to read and diff than JSON for large documents.

---

## Files to Create/Modify

...

```
src/
├── docs/
│ └── openapi.yaml # New — full OpenAPI spec
├── config/
│ └── swagger.ts # New — serve the spec via Express
└── app.ts # Mount the docs route
```

...

---

## `src/docs/openapi.yaml`

```yaml

openapi: "3.0.3"

info:

title: Task Management API

version: "1.0.0"

description: |

Multi-tenant SaaS task management API.

All organization-scoped endpoints require membership in that organization.

Role hierarchy: OWNER ADMIN MEMBER VIEWER.

servers:

- url: http://localhost:3000/api/v1

description: Local development

SECURITY SCHEME

components:

securitySchemes:

bearerAuth:

type: http

scheme: bearer

bearerFormat: JWT

REUSABLE SCHEMAS

schemas:

— Primitives — _____

TaskStatus:

type: string

enum: [TODO, IN_PROGRESS, DONE]

TaskPriority:
type: string
enum: [LOW, MEDIUM, HIGH]

OrgRole:
type: string
enum: [OWNER, ADMIN, MEMBER, VIEWER]

— Shared objects —————

UserSummary:
type: object
properties:
 id:
 type: string
 example: clxyz123
 firstName:
 type: string
 example: Jane
 lastName:
 type: string
 example: Doe
 email:
 type: string
 format: email
 example: jane@example.com

Pagination:
type: object
properties:
 total:
 type: integer
 example: 42
 limit:
 type: integer
 example: 20
 offset:
 type: integer
 example: 0
 hasNextPage:
 type: boolean
 example: true

hasPreviousPage:
type: boolean
example: false

— Task

Task:

type: object
properties:
id:
type: string
example: cltask789
title:
type: string
example: Write API documentation
description:
type: string
nullable: true
example: Cover all endpoints with examples
status:
\$ref: "#/components/schemas/TaskStatus"
priority:
\$ref: "#/components/schemas/TaskPriority"
dueDate:
type: string
format: date-time
nullable: true
example: "2024-03-01T00:00:00.000Z"
deletedAt:
type: string
format: date-time
nullable: true
example: null
organizationId:
type: string
example: clorg456
createdAt:
type: string
format: date-time
example: "2024-01-15T12:00:00.000Z"
updatedAt:
type: string

format: date-time
example: "2024-01-15T12:00:00.000Z"
creator:
 \$ref: "#/components/schemas/UserSummary"
assignee:
 nullable: true
 allOf:
 - \$ref: "#/components/schemas/UserSummary"

— Error responses —————

ErrorResponse:
 type: object
 required: [status, code, message]
 properties:
 status:
 type: string
 enum: [error]
 code:
 type: string
 example: VALIDATION_ERROR
 message:
 type: string
 example: Validation failed
 errors:
 type: object
 additionalProperties:
 type: array
 items:
 type: string
 example:
 title: ["Title cannot be empty"]
 status: ["Status must be TODO, IN_PROGRESS, or DONE"]

—————

REUSABLE PARAMETERS

—————

parameters:

organizationId:
 name: organizationId

in: path
required: true
schema:
 type: string
description: The ID of the organization
example: clorg456

taskId:
 name: taskId
 in: path
 required: true
 schema:
 type: string
 description: The ID of the task
 example: cltask789

limit:
 name: limit
 in: query
 schema:
 type: integer
 minimum: 1
 maximum: 100
 default: 20
 description: Number of results to return

offset:
 name: offset
 in: query
 schema:
 type: integer
 minimum: 0
 default: 0
 description: Number of results to skip

REUSABLE RESPONSES

responses:

Unauthorized:

description: Missing or invalid JWT token

content:

application/json:

schema:

\$ref: "#/components/schemas/ErrorResponse"

examples:

missingToken:

summary: No Authorization header

value:

status: error

code: MISSING_TOKEN

message: Authorization header missing or malformed

expiredToken:

summary: Token has expired

value:

status: error

code: TOKEN_EXPIRED

message: Token has expired

Forbidden:

description: Authenticated but insufficient role

content:

application/json:

schema:

\$ref: "#/components/schemas/ErrorResponse"

example:

status: error

code: INSUFFICIENT_ROLE

message: This action requires MEMBER role or higher

NotFound:

description: Resource not found or not accessible

content:

application/json:

schema:

\$ref: "#/components/schemas/ErrorResponse"

example:

status: error

code: TASK_NOT_FOUND

message: Task not found

ValidationError:

description: Request body or query params failed validation

content:

application/json:

schema:

\$ref: "#/components/schemas/ErrorResponse"

example:

status: error

code: VALIDATION_ERROR

message: Validation failed

errors:

title: ["Title cannot be empty"]

#

GLOBAL SECURITY DEFAULT

#

security:

- bearerAuth: []

#

PATHS

#

paths:

— Auth

/auth/register:

post:

tags: [Auth]

summary: Register a new user account

security: [] # Public endpoint – overrides global security

requestBody:

required: true

content:

application/json:

schema:

type: object

required: [email, password, firstName, lastName]

properties:

email:

type: string
format: email
example: jane@example.com
password:
type: string
minLength: 8
maxLength: 72
description: Must contain at least one uppercase letter and one number
example: SecurePass1
firstName:
type: string
maxLength: 100
example: Jane
lastName:
type: string
maxLength: 100
example: Doe

responses:

"201":

description: User registered successfully

content:

application/json:

schema:

type: object

properties:

status:

type: string

enum: [success]

data:

type: object

properties:

user:

type: object

properties:

id:

type: string

email:

type: string

firstName:

type: string

lastName:

type: string

createdAt:
 type: string
 format: date-time
 token:
 type: string
 description: JWT bearer token
"409":
 description: Email already registered
 content:
 application/json:
 schema:
 \$ref: "#/components/schemas/ErrorResponse"
 example:
 status: error
 code: EMAIL_TAKEN
 message: An account with this email already exists
"422":
 \$ref: "#/components/responses/ValidationError"

/auth/login:

 post:
 tags: [Auth]
 summary: Log in and receive a JWT token
 security: []
 requestBody:
 required: true
 content:
 application/json:
 schema:
 type: object
 required: [email, password]
 properties:
 email:
 type: string
 format: email
 example: jane@example.com
 password:
 type: string
 example: SecurePass1
 responses:
 "200":
 description: Login successful

content:
 application/json:
 schema:
 type: object
 properties:
 status:
 type: string
 enum: [success]
 data:
 type: object
 properties:
 token:
 type: string
 user:
 \$ref: "#/components/schemas/UserSummary"

"401":

 description: Invalid credentials
 content:
 application/json:
 schema:
 \$ref: "#/components/schemas/ErrorResponse"
 example:
 status: error
 code: INVALID_CREDENTIALS
 message: Invalid email or password

"422":

 \$ref: "#/components/responses/ValidationError"

— Organizations

/organizations:

 post:

 tags: [Organizations]

 summary: Create a new organization

 description: The authenticated user becomes the OWNER of the new organization.

 requestBody:

 required: true

 content:

 application/json:

 schema:

 type: object

required: [name, slug]
properties:
 name:
 type: string
 maxLength: 255
 example: Acme Corporation
 slug:
 type: string
 maxLength: 100
 pattern: "[a-z0-9]+(?:-[a-z0-9]+)*\$"
 example: acme-corp

responses:

"201":

description: Organization created

content:

 application/json:

 schema:

 type: object

 properties:

 status:

 type: string

 enum: [success]

 data:

 type: object

 properties:

 organization:

 type: object

 properties:

 id:

 type: string

 name:

 type: string

 slug:

 type: string

 createdAt:

 type: string

 format: date-time

"401":

 \$ref: "#/components/responses/Unauthorized"

"409":

description: Slug already taken

content:

application/json:
schema:
 \$ref: "#/components/schemas/ErrorResponse"
example:
 status: error
 code: SLUG_TAKEN
 message: The slug "acme-corp" is already taken
"422":
 \$ref: "#/components/responses/ValidationError"

/organizations/{organizationId}/members:

post:

tags: [Organizations]

summary: Add a user to the organization

description: |

Requires ADMIN or OWNER role.

ADMIN can assign MEMBER or VIEWER roles only.

OWNER can assign ADMIN, MEMBER, or VIEWER roles.

parameters:

- \$ref: "#/components/parameters/organizationId"

requestBody:

required: true

content:

application/json:

schema:

type: object

required: [email]

properties:

email:

 type: string

 format: email

 example: bob@example.com

role:

 \$ref: "#/components/schemas/OrgRole"

 description: Defaults to MEMBER. OWNER cannot be assigned.

responses:

"201":

description: Member added

content:

application/json:

schema:

 type: object

properties:
status:
 type: string
 enum: [success]
data:
 type: object
 properties:
 membership:
 type: object
 properties:
 user:
 \$ref: "#/components/schemas/UserSummary"
 role:
 \$ref: "#/components/schemas/OrgRole"
 joinedAt:
 type: string
 format: date-time

"401":
 \$ref: "#/components/responses/Unauthorized"

"403":
 \$ref: "#/components/responses/Forbidden"

"404":
description: Organization not found or user not registered
content:
 application/json:
 schema:
 \$ref: "#/components/schemas/ErrorResponse"

"409":
description: User is already a member
content:
 application/json:
 schema:
 \$ref: "#/components/schemas/ErrorResponse"
 example:
 status: error
 code: ALREADY_A_MEMBER
 message: This user is already a member of the organization

"422":
 \$ref: "#/components/responses/ValidationError"

— Tasks

/organizations/{organizationId}/tasks:

get:

tags: [Tasks]

summary: List tasks in an organization

description: Requires VIEWER role or higher.

parameters:

- \$ref: "#/components/parameters/organizationId"

- \$ref: "#/components/parameters/limit"

- \$ref: "#/components/parameters/offset"

- name: status

in: query

schema:

\$ref: "#/components/schemas/TaskStatus"

description: Filter by task status

- name: assigneeId

in: query

schema:

type: string

description: Filter by assignee user ID

- name: dueDateFrom

in: query

schema:

type: string

format: date-time

description: Filter tasks with dueDate on or after this datetime (ISO 8601)

- name: dueDateTo

in: query

schema:

type: string

format: date-time

description: Filter tasks with dueDate on or before this datetime (ISO 8601)

responses:

"200":

description: Paginated list of tasks

content:

application/json:

schema:

type: object

```
properties:
  status:
    type: string
    enum: [success]
  data:
    type: object
    properties:
      tasks:
        type: array
        items:
          $ref: "#/components/schemas/Task"
      pagination:
        $ref: "#/components/schemas/Pagination"
"401":
  $ref: "#/components/responses/Unauthorized"
"403":
  $ref: "#/components/responses/Forbidden"
"404":
  $ref: "#/components/responses/NotFound"
"422":
  $ref: "#/components/responses/ValidationError"
```

```
post:
  tags: [Tasks]
  summary: Create a task in an organization
  description: Requires MEMBER role or higher.
  parameters:
    - $ref: "#/components/parameters/organizationId"
  requestBody:
    required: true
    content:
      application/json:
        schema:
          type: object
          required: [title]
          properties:
            title:
              type: string
              maxLength: 500
              example: Write API documentation
          description:
            type: string
```

maxLength: 10000
nullable: true
example: Cover all endpoints with examples
status:
 \$ref: "#/components/schemas/TaskStatus"
 default: TODO
priority:
 \$ref: "#/components/schemas/TaskPriority"
 default: MEDIUM
assigneeId:
 type: string
 nullable: true
 description: Must be a member of this organization
 example: clxyz789
dueDate:
 type: string
 format: date-time
 nullable: true
 example: "2024-03-01T00:00:00.000Z"

responses:

"201":

 description: Task created

 content:

 application/json:

 schema:

 type: object

 properties:

 status:

 type: string

 enum: [success]

 data:

 type: object

 properties:

 task:

 \$ref: "#/components/schemas/Task"

"401":

 \$ref: "#/components/responses/Unauthorized"

"403":

 \$ref: "#/components/responses/Forbidden"

"404":

 \$ref: "#/components/responses/NotFound"

"422":

\$ref: "#/components/responses/ValidationError"

/organizations/{organizationId}/tasks/{taskId}:

get:

tags: [Tasks]

summary: Get a task by ID

description: Requires VIEWER role or higher.

parameters:

- \$ref: "#/components/parameters/organizationId"
- \$ref: "#/components/parameters/taskId"

responses:

"200":

description: Task found

content:

application/json:

schema:

type: object

properties:

status:

type: string

enum: [success]

data:

type: object

properties:

task:

\$ref: "#/components/schemas/Task"

"401":

\$ref: "#/components/responses/Unauthorized"

"403":

\$ref: "#/components/responses/Forbidden"

"404":

\$ref: "#/components/responses/NotFound"

patch:

tags: [Tasks]

summary: Update a task

description: |

Requires MEMBER role or higher.

Only include fields you want to change.

Send `assigneeId: null` to explicitly unassign.

Omit `assigneeId` entirely to leave the current assignee unchanged.

parameters:

- \$ref: "#/components/parameters/organizationId"
- \$ref: "#/components/parameters/taskId"

requestBody:

required: true

content:

application/json:

schema:

type: object

minProperties: 1

properties:

title:

type: string

maxLength: 500

example: Write API documentation (revised)

description:

type: string

maxLength: 10000

nullable: true

example: Cover all endpoints with examples and edge cases

status:

\$ref: "#/components/schemas/TaskStatus"

priority:

\$ref: "#/components/schemas/TaskPriority"

assigneeId:

type: string

nullable: true

description: Pass null to unassign. Omit to leave unchanged.

example: clxyz789

dueDate:

type: string

format: date-time

nullable: true

example: "2024-04-01T00:00:00.000Z"

responses:

"200":

description: Task updated

content:

application/json:

schema:

type: object

properties:

status:

```
    type: string
    enum: [success]
  data:
    type: object
    properties:
      task:
        $ref: "#/components/schemas/Task"
"401":
  $ref: "#/components/responses/Unauthorized"
"403":
  $ref: "#/components/responses/Forbidden"
"404":
  $ref: "#/components/responses/NotFound"
"422":
  $ref: "#/components/responses/ValidationError"
```

delete:

tags: [Tasks]

summary: Soft-delete a task

description: |

Requires MEMBER role or higher.

Sets `deletedAt` timestamp. The task is excluded from list/get responses automatically. Hard deletion is not exposed via the API.

parameters:

- \$ref: "#/components/parameters/organizationId"

- \$ref: "#/components/parameters/taskId"

responses:

"200":

description: Task soft-deleted

content:

application/json:

schema:

type: object

properties:

status:

type: string

enum: [success]

data:

type: object

properties:

task:

\$ref: "#/components/schemas/Task"

```

    "401":
      $ref: "#/components/responses/Unauthorized"
    "403":
      $ref: "#/components/responses/Forbidden"
    "404":
      $ref: "#/components/responses/NotFound"
  ...

---

## `src/config/swagger.ts`

``typescript
import { Express, Request, Response } from "express";
import swaggerUi from "swagger-ui-express";
import { readFileSync } from "fs";
import { load } from "js-yaml";
import path from "path";

export function mountDocs(app: Express): void {
  const specPath = path.resolve(__dirname, "../docs/openapi.yaml");
  const spec = load(readFileSync(specPath, "utf8")) as Recordstring, unknown;

  app.use(
    "/api/docs",
    swaggerUi.serve,
    swaggerUi.setup(spec, {
      customSiteTitle: "Task Management API Docs",
      swaggerOptions: {
        persistAuthorization: true, // keeps the JWT filled in across page refreshes
        filter: true, // enables the tag/search filter bar
        displayRequestDuration: true, // shows response time in the UI
      },
    })
  );

  // Expose the raw spec for tooling (Postman import, code-gen, etc.)
  app.get("/api/docs/spec.json", (_req: Request, res: Response) => {
    res.json(spec);
  });
}
...

```

****Why `js-yaml` instead of the `yaml` package listed in the install step:****

`js-yaml` is already a transitive dependency of many Express projects and its `load()` API is stable. If your project already has `yaml` installed, replace the import with:

```
``typescript
import { parse } from "yaml";
const spec = parse(readFileSync(specPath, "utf8")) as Recordstring, unknown;
``
```

Both work identically here. Pick whichever is already in your `package.json` to avoid a redundant dependency.

```
## `src/app.ts` – mount the docs route
```

Add one import and one call. Everything else in `app.ts` stays the same.

```
``typescript
// --- existing imports ---
import { mountDocs } from "./config/swagger";

// --- after all route registrations, before the error handler ---
mountDocs(app);
``
```

Placing it after routes but before the error handler ensures the docs UI itself is never caught by your 404 fallback.

```
## Verification
```

Start the server and open:

```
| URL | What you see |
|--|--|
| `http://localhost:3000/api/docs` | Interactive Swagger UI |
| `http://localhost:3000/api/docs/spec.json` | Raw spec as JSON |
```

To authenticate inside the UI:

1. Call `POST /auth/login` from the UI.
2. Copy the `token` value from the response.
3. Click **Authorize** (top right), paste the token, click **Authorize**.
4. All subsequent requests from the UI will include `Authorization: Bearer token`.

Optional / Future Enhancements

- **Spec validation in CI** – run `npx @redocly/cli lint src/docs/openapi.yaml` in your pipeline to catch broken `\$ref`s and missing required fields before they reach production.
- **ReDoc alternative** – `redoc-express` renders a cleaner read-only view; useful for public-facing docs alongside the Swagger UI for internal testing.
- **Response validation middleware** – `express-openapi-validator` can validate actual API responses against the spec at runtime in development, surfacing drift between code and docs immediately.
- **Versioned spec** – if a v2 API is introduced, the spec can be split into `openapi.v1.yaml` / `openapi.v2.yaml` and served at `/api/v1/docs` and `/api/v2/docs` respectively.

USER: Review our entire project architecture and identify any inconsistencies or areas where we deviated from the patterns we established early on. Suggest refactorings to make the codebase more consistent.

Note: Specific factual details should be independently verified. If uncertainty exists, consult authoritative sources.

Architecture Review

This review covers every pattern established in the early setup steps and checks each subsequent addition for consistency. Findings are grouped by severity.

1. Inconsistencies Found

1.1 `delete` endpoint returns 200 with body – contradicts earlier decision

In the task CRUD step, the delete controller was specified to return `204` with no body:

```
``typescript
// Specified in task CRUD step
res.status(204).send();
``
```

The OpenAPI spec documents `DELETE /tasks/:taskId` as returning `200` with a task body:

```
``yaml
# In openapi.yaml
delete:
  responses:
    "200":
      description: Task soft-deleted
      content:
        application/json:
          schema:
            properties:
              task:
                $ref: "#/components/schemas/Task"
``
```

These contradict each other. The controller returns 204 with no body. The spec documents 200 with a body. One of them is wrong.

****Resolution – pick one and apply it consistently:****

Option A (keep 204): Update the spec to document 204 with no body. This is correct for a delete operation per HTTP semantics.

Option B (return 200 with body): Update the controller to return the soft-deleted task. This is defensible for soft deletes specifically – the client receives confirmation of what was deleted and can see the `deletedAt` timestamp.

For soft deletes, Option B is more useful to clients. The `tasksService.delete` method already fetches the task before soft-deleting it – returning it costs nothing extra.

****Patch – `src/modules/tasks/tasks.controller.ts`:****

```
``typescript
async delete(req: OrgScopedRequest, res: Response, next: NextFunction): Promisevoid {
  try {
    const task = await tasksService.delete(
```

```

    req.params.taskId,
    req.membership.organizationId,
    req.user.sub
  );
  res.status(200).json({ status: "success", data: { task } });
} catch (err) {
  next(err);
}
},
...

```

****Patch — `src/modules/tasks/tasks.service.ts` — `delete` method return value:****

```

``typescript
async delete(taskId: string, organizationId: string, actorId: string) {
  const task = await tasksService.getById(taskId, organizationId);

  const deleted = await prisma.task.update({
    where: { id: taskId },
    data: { deletedAt: new Date() },
    select: TASK_SELECT,
  });

  auditService
    .log({ ... })
    .catch(...);

  return deleted; // Return the soft-deleted task
},
...

```

****Patch — `src/docs/openapi.yaml`:**** The spec already documents 200 with a body, so no change needed there if Option B is chosen.

1.2 `requireRole` default changed without updating all call sites

In the initial `requireRole` implementation, the default minimum role was `MEMBER`. After the VIEWER role was added, the default was changed to `VIEWER`. This is correct, but the `organizations.routes.ts` `addMember` route does not pass an explicit role:

```

```typescript
// src/modules/organizations/organizations.routes.ts
router.post(
 "/:organizationId/members",
 authenticate,
 requireRole("ADMIN"), // - explicit - correct
 ...
);
```

```

This one is fine. The issue is in the `requireOrgMembership` alias suggested in the "reusable middleware" step:

```

```typescript
export const requireOrgMembership = requireRole("VIEWER");
```

```

This alias was introduced but never actually used in any route file – the routes all use `requireRole("VIEWER")` directly. The alias is dead code if it was added to `requireRole.ts`. Either use it consistently or remove it.

****Resolution:**** Remove the alias. `requireRole("VIEWER")` is already readable. Dead exports create confusion about what is actually in use.

****Patch – `src/middleware/requireRole.ts`:**** Remove the `requireOrgMembership` export if it was added.

1.3 `validate` middleware writes back to `req[segment]` but controllers cast with `as`

After the validation middleware step, controllers use:

```

```typescript
req.body as CreateTaskInput
req.query as unknown as ListTasksQuery
```

```

The `validate` middleware writes `result.data` back to `req[segment]`, so the cast is safe at runtime. But the double cast `as unknown as ListTasksQuery` on `req.query` is a code smell – it signals that TypeScript does not believe the type is correct, which it is not without augmenting the `Request` type.

The root cause: Express types `req.query` as `ParsedQs`, not as the validated schema type. Writing `result.data` back to `req.query` does not change the TypeScript type.

Resolution: Augment the `Request` type to carry the parsed query, or accept the cast as a known limitation and document it. The cast is safe because `validate` runs before the controller. The `as unknown as` pattern is the standard workaround for this Express/TypeScript limitation.

No code change is strictly required, but adding a comment at the cast site prevents future developers from removing it thinking it is unnecessary:

```
```typescript
// req.query is typed as ParsedQs by Express. The validate middleware
// has already parsed and transformed it to ListTasksQuery via Zod.
// The cast is safe – validate runs before this handler.
const { tasks, pagination } = await tasksService.list(
 req.membership.organizationId,
 req.query as unknown as ListTasksQuery
);
```
```

1.4 `authService.login` throws `ConflictError` for invalid credentials

In `auth.service.ts`:

```
```typescript
throw new ConflictError("Invalid email or password", "INVALID_CREDENTIALS");
```
```

`ConflictError` maps to HTTP 409. Invalid credentials should be `UnauthorizedError` (401). This was noted as correct in the login step but the wrong class was used. The `auth.validation.ts` step introduced `loginSchema` with the correct error class in the controller, but the service still uses `ConflictError`.

Patch – `src/modules/auth/auth.service.ts`:

```
```typescript
// Before
throw new ConflictError("Invalid email or password", "INVALID_CREDENTIALS");
```

```
// After
throw new UnauthorizedError("Invalid email or password", "INVALID_CREDENTIALS");
...
```

---

### 1.5 `ForbiddenError` is imported in `organizations.service.ts` but not defined in the import path shown

In the `addMember` service method:

```
```typescript
throw new ForbiddenError(
  "Admins cannot assign the ADMIN role. Only owners can.",
  "INSUFFICIENT_ROLE"
);
```
```

The import for `ForbiddenError` was not shown in the `organizations.service.ts` file in the add-member step. The file imports `ConflictError` and `NotFoundError` but the `ForbiddenError` import was added inline in the role-assignment guard without updating the import statement.

\*\*Patch — `src/modules/organizations/organizations.service.ts`:\*\*

```
```typescript
// Before
import { ConflictError, NotFoundError } from "../../errors/HttpErrors";

// After
import { ConflictError, NotFoundError, ForbiddenError } from "../../errors/HttpErrors";
```
```

---

### 1.6 `DUMMY\_HASH` in `auth.service.ts` may not be a valid bcrypt hash

The timing-attack prevention in `authService.login` uses:

```
```typescript
const DUMMY_HASH =
  "$2b$12$invalidhashpaddinginvalidhashpaddinginvalidhashpadXXXXX";
```
```

`bcrypt.compare` with an invalid hash string throws rather than returning `false` in some versions of `bcryptjs`. If it throws, the error propagates to the controller as an unhandled error, producing a 500 instead of a 401 – and the timing protection fails because the error path is faster than a real bcrypt comparison.

**\*\*Resolution:\*\*** Use a pre-computed valid bcrypt hash as the dummy. Generate one once:

```
``typescript
// Run this once in a script or REPL to generate the value:
// const bcrypt = require("bcryptjs");
// console.log(bcrypt.hashSync("dummy-password-for-timing", 12));
````
```

Then hardcode the output:

```
``typescript
// A valid bcrypt hash – bcrypt.compare will run the full comparison
// and return false, rather than throwing on an invalid hash format.
const DUMMY_HASH =
  "$2b$12$LQv3c1yqBwVHxkd0LHAKCOYz6TtxMQJqhN8/LewdBPj/RK.s5uR.m";
````
```

**\*\*Patch – `src/modules/auth/auth.service.ts`:\*\*** Replace the `DUMMY\_HASH` constant with a pre-computed valid hash.

---

### 1.7 `swagger.ts` imports `js-yaml` but the install step specified `yaml`

The install step:

```
``bash
npm install swagger-ui-express yaml
````
```

The `swagger.ts` implementation:

```
``typescript
import { load } from "js-yaml";
````
```

`js-yaml` and `yaml` are different packages with different APIs. The implementation uses `js-yaml` but the install step installs `yaml`. This will fail at runtime with a module-not-found error unless `js-yaml` is already a transitive dependency.

**\*\*Resolution:\*\*** Make the install step and the implementation consistent. The implementation already notes this discrepancy and provides the `yaml` alternative. Apply it:

**\*\*Patch — `src/config/swagger.ts`:\*\***

```
``typescript
// Before
import { load } from "js-yaml";
const spec = load(readFileSync(specPath, "utf8")) as Recordstring, unknown;

// After
import { parse } from "yaml";
const spec = parse(readFileSync(specPath, "utf8")) as Recordstring, unknown;
...

```

---

## ## 2. Pattern Consistency Audit

### ### 2.1 Fire-and-forget pattern — consistent ✓

Every async side effect (audit log, notification) uses the same pattern:

```
``typescript
service.method(...).catch((err) = {
 console.error("Descriptive label", { context, error: err });
});
...

```

Applied consistently in `tasksService.create`, `update`, and `delete`. No deviations.

---

### ### 2.2 `TASK\_SELECT` constant — consistent ✓

The `TASK\_SELECT` constant is defined once in `tasks.service.ts` and used in every Prisma query that returns a task. `deletedAt` was correctly added to `TASK\_SELECT` in the soft-delete step. No deviations.

---

### ### 2.3 Validation middleware placement — consistent ✓

Every route that accepts a body or query params applies `validate(schema)` or `validate(schema, "query")` before the controller. The order is always: `authenticate` → `requireRole` → `validate` → controller. No deviations.

---

### ### 2.4 Error class usage — one deviation (covered in 1.4 above)

All other error throws use the correct subclass for the HTTP status code. The `ConflictError` for invalid credentials in `auth.service.ts` is the only deviation.

---

### ### 2.5 `@@map` and `@map` on all models — consistent ✓

Every model uses `@@map("snake_case_table_name")` and every multi-word field uses `@map("snake_case_column_name")`. The `Comment` and `Notification` models added later follow the same convention. No deviations.

---

### ### 2.6 `select` on every Prisma query — mostly consistent

Most queries use explicit `select` to avoid returning `passwordHash` and to control response shape. One gap: `prisma.task.delete` in the original hard-delete implementation did not need a `select` because it returned nothing. After the soft-delete refactor, `prisma.task.update` in `delete` should use `TASK_SELECT` — covered in the patch for 1.1.

---

### ### 2.7 `mergeParams: true` — consistent ✓

Both `tasks.routes.ts` and `comments.routes.ts` use `Router({ mergeParams: true })`. The `notifications.routes.ts` does not use `mergeParams` because it is not nested under a parent with params. Correct in all three cases.

---

### ## 3. Summary of Required Patches

| # | File | Change |

|---|---|---|

| 1.1 | `tasks.controller.ts` | `delete` returns 200 with task body |  
| 1.1 | `tasks.service.ts` | `delete` returns the soft-deleted task |  
| 1.2 | `requireRole.ts` | Remove `requireOrgMembership` alias if added |  
| 1.4 | `auth.service.ts` | Replace `ConflictError` with `UnauthorizedError` for invalid credentials |  
| 1.5 | `organizations.service.ts` | Add `ForbiddenError` to imports |  
| 1.6 | `auth.service.ts` | Replace `DUMMY\_HASH` with a pre-computed valid bcrypt hash |  
| 1.7 | `swagger.ts` | Use `yaml` package instead of `js-yaml` |

---

### ## 4. No Changes Needed

The following areas were reviewed and are consistent with established patterns:

- Role hierarchy in `requireRole.ts` – `VIEWER` insertion at index 0 is correct
- `buildDiff` in `auditService` – used consistently in `update` and `delete`
- `assertOrgMember` in `tasksService` – called before both create and update when `assigneId` is present
- `assertActiveTask` in `commentsService` – correctly mirrors `getById` logic without importing `tasksService`
- Pagination shape – `{ total, limit, offset, hasNextPage, hasPreviousPage }` used consistently in tasks, comments, and notifications
- `ACTIVE\_TASK\_FILTER` constant – applied in `list` and `getById`; `assertActiveTask` in comments replicates the same check independently, which is correct given the no-cross-module-import decision