TRANSCRIPT A

Test Environment

Date: Feb 2026
Model: Claude
Interface: Claude API
Continuity Layer: Calmkeep (visit @ Calmkeep.ai)

This transcript was generated using Claude Sonnet 4.6 through the Claude App- It is here for comparison with the Identically prompted Calmkeep via Claude API Transcript. The prompt sequence was identical to the prompts used for Transcript B in the Calmkeep chat interface.

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.

Feb 18

Here's the structure:

```
task-api/
├── src/
│   ├── config/
│   │   ├── database.ts
│   │   └── env.ts
│   ├── middleware/
│   │   ├── auth.ts
│   │   ├── tenant.ts
│   │   └── errorHandler.ts
│   ├── modules/
│   │   ├── auth/
│   │   │   ├── auth.controller.ts
│   │   │   ├── auth.service.ts
│   │   │   └── auth.routes.ts
│   │   ├── tenants/
│   │   │   ├── tenant.controller.ts
│   │   │   ├── tenant.service.ts
│   │   │   └── tenant.routes.ts
│   │   └── tasks/
│   │       ├── task.controller.ts
│   │       ├── task.service.ts
│   │       └── task.routes.ts
│   ├── db/
│   │   ├── migrations/
│   │   └── schema.ts
│   ├── types/
│   │   └── index.ts
│   └── app.ts
├── .env
├── package.json
└── tsconfig.json
```

Why this structure:

`modules/` — feature-based grouping keeps auth, tenants, and tasks self-contained. Each module owns its routes, controller (HTTP layer), and service (business logic).

`middleware/` — `auth.ts` validates JWTs, `tenant.ts` resolves and attaches the tenant from the token so every downstream handler has `req.tenantId` available.

`db/` — migrations live separately from app code; `schema.ts` holds table definitions/query helpers (or your ORM models if you add one).

`config/` — `env.ts` validates and exports all env vars in one place so you fail fast on startup if something's missing.

`types/` — shared TypeScript interfaces like `AuthRequest`, `JWTPayload`, `Task`, `Tenant`.

Init commands:

bash

```bash
mkdir task-api && cd task-api
npm init -y
npm install express pg jsonwebtoken bcrypt dotenv
npm install -D typescript ts-node nodemon @types/express @types/pg @types/jsonwebtoken @types/bcrypt
npx tsc --init
```

tsconfig.json key settings:

json

```json
{
  "compilerOptions": {
    "target": "ES2020",
    "module": "commonjs",
    "rootDir": "./src",
    "outDir": "./dist",
    "strict": true,
    "esModuleInterop": true
  }
}
```

Want me to scaffold the actual code next — `app.ts`, env config, DB connection, or the auth module?

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.

Feb 18

sql

```sql
-- Enable UUID extension
CREATE EXTENSION IF NOT EXISTS "uuid-ossp";

-- Organizations (tenants)
CREATE TABLE organizations (
  id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
  name VARCHAR(255) NOT NULL,
  slug VARCHAR(100) UNIQUE NOT NULL,
  created_at TIMESTAMPTZ DEFAULT NOW(),
  updated_at TIMESTAMPTZ DEFAULT NOW()
);

-- Users
CREATE TABLE users (
  id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
  email VARCHAR(255) UNIQUE NOT NULL,
  password_hash VARCHAR(255) NOT NULL,
  created_at TIMESTAMPTZ DEFAULT NOW(),
  updated_at TIMESTAMPTZ DEFAULT NOW()
);

-- User <-> Organization (many-to-many + role per org)
CREATE TABLE organization_members (
  id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
  user_id UUID NOT NULL REFERENCES users(id) ON DELETE CASCADE,
  org_id UUID NOT NULL REFERENCES organizations(id) ON DELETE CASCADE,
  role VARCHAR(50) NOT NULL DEFAULT 'member', -- 'owner' | 'admin' | 'member'
  joined_at TIMESTAMPTZ DEFAULT NOW(),
  UNIQUE(user_id, org_id)
);

-- Tasks
CREATE TABLE tasks (
  id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
  org_id UUID NOT NULL REFERENCES organizations(id) ON DELETE CASCADE,
  created_by UUID NOT NULL REFERENCES users(id),
  assigned_to UUID REFERENCES users(id) ON DELETE SET NULL,
  title VARCHAR(500) NOT NULL,
  description TEXT,
  status VARCHAR(50) NOT NULL DEFAULT 'todo', -- 'todo' | 'in_progress' | 'done'
  priority VARCHAR(50) NOT NULL DEFAULT 'medium', -- 'low' | 'medium' | 'high'
  due_date TIMESTAMPTZ,
  created_at TIMESTAMPTZ DEFAULT NOW(),
  updated_at TIMESTAMPTZ DEFAULT NOW()
);

-- Indexes
CREATE INDEX idx_org_members_user_id ON organization_members(user_id);
CREATE INDEX idx_org_members_org_id ON organization_members(org_id);
CREATE INDEX idx_tasks_org_id ON tasks(org_id);
CREATE INDEX idx_tasks_assigned_to ON tasks(assigned_to);
CREATE INDEX idx_tasks_status ON tasks(status);
CREATE INDEX idx_tasks_org_status ON tasks(org_id, status);
```

Relationships:

`users` ↔ `organizations` — many-to-many through `organization_members`. A user can belong to multiple orgs, each with their own role. The `UNIQUE(user_id, org_id)` constraint prevents duplicate memberships.

`tasks` → `organizations` — each task belongs to exactly one org via `org_id`. This is your tenant isolation boundary — every query against tasks should always filter by `org_id`.

`tasks` → `users` (x2) — `created_by` is non-nullable (who made it), `assigned_to` is nullable and goes `SET NULL` if the user is deleted rather than cascading.

Multi-tenant isolation pattern — in every task query you'll enforce:

sql

```sql
SELECT * FROM tasks WHERE org_id = $1 AND id = $2;
-- never query tasks without org_id in the WHERE clause
```

The `slug` on organizations is for URL-friendly tenant identification (e.g. `/acme/tasks`).

Want the migrations setup, TypeScript types for these, or move on to the auth module?

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.

Feb 18

prisma

```prisma
// prisma/schema.prisma

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

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

model Organization {
  id        String   @id @default(uuid())
  name      String
  slug      String   @unique
  createdAt DateTime @default(now()) @map("created_at")
  updatedAt DateTime @updatedAt @map("updated_at")

  members   OrganizationMember[]
  tasks     Task[]

  @@map("organizations")
}

model User {
  id           String   @id @default(uuid())
  email        String   @unique
  passwordHash String   @map("password_hash")
  createdAt    DateTime @default(now()) @map("created_at")
  updatedAt    DateTime @updatedAt @map("updated_at")

  memberships   OrganizationMember[]
  createdTasks  Task[]  @relation("TaskCreator")
  assignedTasks Task[]  @relation("TaskAssignee")
  @@map("users")
}

model OrganizationMember {
  id       String   @id @default(uuid())
  userId   String   @map("user_id")
  orgId    String   @map("org_id")
  role     Role     @default(MEMBER)
  joinedAt DateTime @default(now()) @map("joined_at")

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

  @@unique([userId, orgId])
  @@index([userId])
  @@index([orgId])
  @@map("organization_members")
}

model Task {
  id          String     @id @default(uuid())
  orgId       String     @map("org_id")
  createdById String     @map("created_by")
  assignedToId String?   @map("assigned_to")
  title       String
  description String?
  status      TaskStatus @default(TODO)
  priority    Priority   @default(MEDIUM)
  dueDate     DateTime?  @map("due_date")
  createdAt   DateTime   @default(now()) @map("created_at")
  updatedAt   DateTime   @updatedAt @map("updated_at")

  organization Organization @relation(fields: [orgId], references:
[id], onDelete: Cascade)
  createdBy    User         @relation("TaskCreator", fields:
[createdById], references: [id])
  assignedTo   User?        @relation("TaskAssignee", fields:
[assignedToId], references: [id], onDelete: SetNull)

  @@index([orgId])
  @@index([assignedToId])
  @@index([status])
  @@index([orgId, status])
  @@map("tasks")
}

enum Role {
  OWNER
  ADMIN
  MEMBER
}

enum TaskStatus {
  TODO
  IN_PROGRESS
  DONE
}

enum Priority {
  LOW
  MEDIUM
  HIGH
}
```

Adjustments and why:

Enums instead of VARCHAR — `Role`, `TaskStatus`, `Priority` are now Prisma enums. Postgres enforces valid values at the DB level, and you get TypeScript types for free.

`@updatedAt` — Prisma automatically updates this field on every write. No need for triggers or manual updates.

Named relations on Task — `"TaskCreator"` and `"TaskAssignee"` disambiguate the two `User` relations on `Task`. Without names Prisma can't tell them apart.

`@@map` / `@map` — keeps snake_case in the DB, camelCase in your TypeScript code. Best of both worlds.

camelCase field names — `passwordHash` not `password_hash` in Prisma; the `@map` handles the column name.

Install and init:

bash

```bash
npm install prisma @prisma/client
npx prisma init
npx prisma migrate dev --name init
npx prisma generate
```

Singleton client (`src/config/database.ts`):

typescript

```
import { PrismaClient } from '@prisma/client';

const globalForPrisma = global as unknown as { prisma: PrismaClient };

export const prisma =
  globalForPrisma.prisma || new PrismaClient({ log: ['query', 'error'] });

if (process.env.NODE_ENV !== 'production') globalForPrisma.prisma = prisma;
```

The singleton pattern prevents creating a new connection pool on every hot-reload in dev.

Ready to build the auth module next?

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.

Feb 18

typescript

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

import { Request } from 'express';
import { Role } from '@prisma/client';

export interface JWTPayload {
  userId: string;
  email: string;
}

export interface AuthRequest extends Request {
  user?: JWTPayload;
  membership?: {
    orgId: string;
    role: Role;
  };
}
```

typescript

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

export const env = {
  JWT_SECRET: process.env.JWT_SECRET!,
  JWT_EXPIRES_IN: process.env.JWT_EXPIRES_IN || '7d',
  DATABASE_URL: process.env.DATABASE_URL!,
  PORT: process.env.PORT || 3000,
};

// Fail fast on startup
const required = ['JWT_SECRET', 'DATABASE_URL'];
required.forEach((key) => {
  if (!process.env[key]) throw new Error(`Missing required env var:
${key}`);
});
```

typescript

```
// src/middleware/auth.ts

import { Response, NextFunction } from 'express';
import jwt from 'jsonwebtoken';
import { AuthRequest, JWTPayload } from '../types';
import { env } from '../config/env';

export const authenticate = (
  req: AuthRequest,
  res: Response,
  next: NextFunction
) => {
  const authHeader = req.headers.authorization;

  if (!authHeader?.startsWith('Bearer ')) {
    return res.status(401).json({ error: 'Missing or invalid
authorization header' });
  }

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

  try {
    const payload = jwt.verify(token, env.JWT_SECRET) as JWTPayload;
    req.user = payload;
    next();
  } catch (err) {
    if (err instanceof jwt.TokenExpiredError) {
      return res.status(401).json({ error: 'Token expired' });
    }
    return res.status(401).json({ error: 'Invalid token' });
  }
};
```

```typescript
// src/middleware/tenant.ts

import { Response, NextFunction } from 'express';
import { AuthRequest } from '../types';
import { prisma } from '../config/database';

// Resolves org from :orgId param and verifies the user is a member
export const resolveTenant = async (
  req: AuthRequest,
  res: Response,
  next: NextFunction
) => {
  const { orgId } = req.params;

  if (!orgId) {
    return res.status(400).json({ error: 'Missing orgId' });
  }

  try {
    const membership = await prisma.organizationMember.findUnique({
      where: {
        userId_orgId: {
          userId: req.user!.userId,
          orgId,
        },
      },
    });

    if (!membership) {
      return res.status(403).json({ error: 'Access denied to this organization' });
    }

    req.membership = { orgId, role: membership.role };
    next();
  } catch {
    return res.status(500).json({ error: 'Internal server error' });
  }
};
```

```typescript
// src/middleware/requireRole.ts

import { Response, NextFunction } from 'express';
import { Role } from '@prisma/client';
import { AuthRequest } from '../types';

const roleHierarchy: Record<Role, number> = {
  OWNER: 3,
  ADMIN: 2,
  MEMBER: 1,
};

export const requireRole = (minimum: Role) => {
  return (req: AuthRequest, res: Response, next: NextFunction) => {
    const userRole = req.membership?.role;

    if (!userRole || roleHierarchy[userRole] < roleHierarchy[minimum]) {
      return res.status(403).json({ error: `Requires ${minimum} role or higher` });
    }

    next();
  };
};
```

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

import { Request, Response, NextFunction } from 'express';

export const errorHandler = (
  err: Error,
  req: Request,
  res: Response,
  next: NextFunction
) => {
  console.error(err.stack);
  res.status(500).json({ error: 'Internal server error' });
};
```

```typescript
// src/app.ts

import express from 'express';
import { authRoutes } from './modules/auth/auth.routes';
import { taskRoutes } from './modules/tasks/task.routes';
import { errorHandler } from './middleware/errorHandler';
```

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

// Public
app.use('/api/auth', authRoutes);

// Protected
app.use('/api/orgs/:orgId/tasks', taskRoutes);

app.use(errorHandler);

export default app;
```

typescript

~~~~
// src/modules/tasks/task.routes.ts

import { Router } from 'express';
import { authenticate } from '../../middleware/auth';
import { resolveTenant } from '../../middleware/tenant';
import { requireRole } from '../../middleware/requireRole';
import { TaskController } from './task.controller';

const router = Router({ mergeParams: true }); // mergeParams to
access :orgId

// All task routes: must be authenticated + org member
router.use(authenticate, resolveTenant);

router.get('/', TaskController.list);
router.post('/', TaskController.create);
router.patch('/:taskId', TaskController.update);

// Only admins/owners can delete
router.delete('/:taskId', requireRole('ADMIN'),
TaskController.delete);

export { router as taskRoutes };
```

**How it chains together:**
```
Request
  → authenticate       (validates JWT, attaches req.user)
~~~~

```
  → resolveTenant      (checks org membership, attaches
req.membership)
  → requireRole(...)   (optional, checks role hierarchy)
  → controller
```

Key points:

`mergeParams: true` on the task router — without this, `:orgId` from the parent route (`/api/orgs/:orgId/tasks`) is invisible inside the task router.

Role hierarchy — `requireRole('ADMIN')` passes for both `ADMIN` and `OWNER` since it checks the numeric hierarchy, not exact equality.

`resolveTenant` always DB-checks membership — never trust `orgId` from the token alone. A user's membership could be revoked after the token was issued.

Token generation (you'll use this in the auth service):

typescript

```typescript
import jwt from 'jsonwebtoken';
import { env } from '../config/env';
import { JWTPayload } from '../types';

export const signToken = (payload: JWTPayload): string => {
  return jwt.sign(payload, env.JWT_SECRET, { expiresIn:
env.JWT_EXPIRES_IN });
};
```

Ready to build the auth module (register/login endpoints) next?

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.

Feb 18

typescript

```typescript
// src/errors/AppError.ts

export class AppError extends Error {
  constructor(
    public message: string,
    public statusCode: number,
    public code: string,
    public details?: unknown
  ) {
    super(message);
    this.name = this.constructor.name;
    Error.captureStackTrace(this, this.constructor);
  }
}
```

typescript

```
// src/errors/errors.ts

import { AppError } from './AppError';

export class NotFoundError extends AppError {
  constructor(resource = 'Resource', details?: unknown) {
    super(`${resource} not found`, 404, 'NOT_FOUND', details);
  }
}

export class UnauthorizedError extends AppError {
  constructor(message = 'Unauthorized', details?: unknown) {
    super(message, 401, 'UNAUTHORIZED', details);
  }
}

export class ForbiddenError extends AppError {
  constructor(message = 'Forbidden', details?: unknown) {
    super(message, 403, 'FORBIDDEN', details);
  }
}

export class ValidationError extends AppError {
  constructor(message = 'Validation failed', details?: unknown) {
    super(message, 422, 'VALIDATION_ERROR', details);
  }
}

export class ConflictError extends AppError {
  constructor(message = 'Conflict', details?: unknown) {
    super(message, 409, 'CONFLICT', details);
  }
}

export class BadRequestError extends AppError {
  constructor(message = 'Bad request', details?: unknown) {
    super(message, 400, 'BAD_REQUEST', details);
  }
}

export class InternalError extends AppError {
  constructor(message = 'Internal server error', details?: unknown) {
    super(message, 500, 'INTERNAL_ERROR', details);
  }
}
```

typescript

```typescript
// src/errors/index.ts

export { AppError } from './AppError';
export * from './errors';
```

typescript

```typescript
// src/types/index.ts  (add to existing file)

export interface ErrorResponse {
  error: {
    code: string;
    message: string;
    details?: unknown;
    requestId?: string;
    timestamp: string;
  };
}
```

typescript

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

import { Request, Response, NextFunction } from 'express';
import { Prisma } from '@prisma/client';
import { AppError, ConflictError, NotFoundError, ValidationError }
from '../errors';
import { ErrorResponse } from '../types';

const isProd = process.env.NODE_ENV === 'production';

const buildErrorResponse = (
  code: string,
  message: string,
  details?: unknown,
  requestId?: string
): ErrorResponse => ({
  error: {
    code,
    message,
    details: isProd ? undefined : details,
    requestId,
    timestamp: new Date().toISOString(),
  },
});

const handlePrismaError = (err:
Prisma.PrismaClientKnownRequestError): AppError => {
  switch (err.code) {
    case 'P2002':
      return new ConflictError(
        `Duplicate value on field: ${(err.meta?.target as
string[])?.join(', ')}`
      );
    case 'P2025':
      return new NotFoundError('Record');
    case 'P2003':
      return new ValidationError('Foreign key constraint failed',
err.meta);
    case 'P2014':
      return new ValidationError('Relation violation', err.meta);
    default:
      return new AppError('Database error', 500, 'DB_ERROR',
err.meta);
  }
};

export const errorHandler = (
  err: Error,
  req: Request,
  res: Response,
  next: NextFunction
) => {
  const requestId = req.headers['x-request-id'] as string |
undefined;

  // Known app errors
  if (err instanceof AppError) {
    return res
      .status(err.statusCode)
      .json(buildErrorResponse(err.code, err.message, err.details,
requestId));
  }

  // Prisma known errors
  if (err instanceof Prisma.PrismaClientKnownRequestError) {
    const appError = handlePrismaError(err);
    return res
```

```
      .status(appError.statusCode)
      .json(buildErrorResponse(appError.code, appError.message,
appError.details, requestId));
  }

  // Prisma validation errors
  if (err instanceof Prisma.PrismaClientValidationError) {
    return res
      .status(422)
      .json(buildErrorResponse('VALIDATION_ERROR', 'Invalid data
provided', undefined, requestId));
  }

  // JWT errors (if not caught in middleware)
  if (err.name === 'JsonWebTokenError') {
    return res
      .status(401)
      .json(buildErrorResponse('INVALID_TOKEN', 'Invalid token',
undefined, requestId));
  }

  if (err.name === 'TokenExpiredError') {
    return res
      .status(401)
      .json(buildErrorResponse('TOKEN_EXPIRED', 'Token expired',
undefined, requestId));
  }

  // Unknown errors — don't leak internals in prod
  console.error('[Unhandled Error]', err);

  return res.status(500).json(
    buildErrorResponse(
      'INTERNAL_ERROR',
      'Internal server error',
      isProd ? undefined : err.message,
      requestId
    )
  );
};
```

typescript

```
// src/middleware/notFound.ts

import { Request, Response } from 'express';
```

```
export const notFound = (req: Request, res: Response) => {
  res.status(404).json({
    error: {
      code: 'ROUTE_NOT_FOUND',
      message: `Route ${req.method} ${req.path} not found`,
      timestamp: new Date().toISOString(),
    },
  });
};
```

typescript

```
// src/app.ts (updated)

import express from 'express';
import { authRoutes } from './modules/auth/auth.routes';
import { taskRoutes } from './modules/tasks/task.routes';
import { errorHandler } from './middleware/errorHandler';
import { notFound } from './middleware/notFound';

const app = express();
app.use(express.json());

app.use('/api/auth', authRoutes);
app.use('/api/orgs/:orgId/tasks', taskRoutes);

// Order matters — notFound before errorHandler
app.use(notFound);
app.use(errorHandler);

export default app;
```

Usage in controllers/services:

typescript

```
import { NotFoundError, ForbiddenError, ConflictError } from '../../errors';

// In a service
const task = await prisma.task.findUnique({ where: { id, orgId } });
if (!task) throw new NotFoundError('Task');

// In middleware
if (!membership) throw new ForbiddenError('Access denied to this organization');
```

```
// Duplicate email on register
throw new ConflictError('Email already in use');
```

What every error response looks like:

json

```
{
  "error": {
    "code": "NOT_FOUND",
    "message": "Task not found",
    "requestId": "abc-123",
    "timestamp": "2024-01-15T10:30:00.000Z"
  }
}
```

Key points:

Prisma error mapping — P2002 (unique constraint), P2025 (record not found), P2003 (foreign key) are the ones you'll hit constantly. They map to clean HTTP responses instead of leaking DB internals.

`details` stripped in prod — stack traces and meta never reach the client in production.

`requestId` — pass `x-request-id` from your gateway/client and it echoes back, making log correlation trivial.

Throw anywhere, catch once — services and controllers just throw, the handler at the bottom catches everything.

Ready for the auth module (register/login) next?

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.

Feb 18

typescript

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

import { ValidationError } from '../../errors';

interface RegisterInput {
  email: string;
  password: string;
}
```

```
export const validateRegisterInput = (body: unknown): RegisterInput => {
  const { email, password } = body as Record<string, unknown>;

  const errs: string[] = [];

  if (!email || typeof email !== 'string') {
    errs.push('Email is required');
  } else if (!/^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(email)) {
    errs.push('Invalid email format');
  }

  if (!password || typeof password !== 'string') {
    errs.push('Password is required');
  } else if (password.length < 8) {
    errs.push('Password must be at least 8 characters');
  } else if (!/(?=.*[a-z])(?=.*[A-Z])(?=.*\d)/.test(password)) {
    errs.push('Password must contain uppercase, lowercase, and a number');
  }

  if (errs.length) throw new ValidationError('Invalid input', errs);

  return { email: email.toLowerCase().trim(), password };
};
```

typescript

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

import bcrypt from 'bcrypt';
import jwt from 'jsonwebtoken';
import { prisma } from '../../config/database';
import { env } from '../../config/env';
import { ConflictError } from '../../errors';
import { JWTPayload } from '../../types';

const SALT_ROUNDS = 12;

export const signToken = (payload: JWTPayload): string => {
  return jwt.sign(payload, env.JWT_SECRET, { expiresIn: env.JWT_EXPIRES_IN });
};

export const registerUser = async (email: string, password: string) => {
  const existing = await prisma.user.findUnique({ where: { email }
});
  if (existing) throw new ConflictError('Email already in use');

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

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

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

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

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

import { Request, Response, NextFunction } from 'express';
import { validateRegisterInput } from './auth.validation';
import { registerUser } from './auth.service';

export const AuthController = {
  async register(req: Request, res: Response, next: NextFunction) {
    try {
      const { email, password } = validateRegisterInput(req.body);
      const { user, token } = await registerUser(email, password);

      res.status(201).json({ user, token });
    } catch (err) {
      next(err);
    }
  },
};
```

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

import { Router } from 'express';
import { AuthController } from './auth.controller';
```

```
const router = Router();

router.post('/register', AuthController.register);

export { router as authRoutes };
```

Example responses:

Success `201`:

json

```json
{
  "user": {
    "id": "uuid",
    "email": "jane@example.com",
    "createdAt": "2024-01-15T10:30:00.000Z"
  },
  "token": "eyJhbGci..."
}
```

Duplicate email `409`:

json

```json
{
  "error": {
    "code": "CONFLICT",
    "message": "Email already in use",
    "timestamp": "2024-01-15T10:30:00.000Z"
  }
}
```

Validation failure `422`:

json

```json
{
  "error": {
    "code": "VALIDATION_ERROR",
    "message": "Invalid input",
    "details": ["Password must contain uppercase, lowercase, and a number"],
    "timestamp": "2024-01-15T10:30:00.000Z"
  }
}
```

Key points:

`select` on create — never return `passwordHash` from any query. Explicitly select only the fields you need.

`SALT_ROUNDS = 12` — bcrypt default is 10; 12 is a reasonable bump for auth endpoints where you can afford ~300ms hashing time.

Timing-safe duplicate check — checking existence before insert is fine here since the unique constraint on `email` will also catch races at the DB level, which Prisma maps to `ConflictError` via our P2002 handler.

Validation is pure — `validateRegisterInput` throws synchronously with all errors collected at once (not fail-fast), so the client gets every problem in one response.

Ready for the login endpoint next?

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

Feb 18

typescript

```
// src/modules/auth/auth.validation.ts (add to existing file)

export const validateLoginInput = (body: unknown): { email: string;
password: string } => {
  const { email, password } = body as Record<string, unknown>;
  const errs: string[] = [];

  if (!email || typeof email !== 'string') errs.push('Email is
required');
  if (!password || typeof password !== 'string') errs.push('Password
is required');

  if (errs.length) throw new ValidationError('Invalid input', errs);

  return { email: (email as string).toLowerCase().trim(), password:
password as string };
};
```

typescript

```
// src/modules/auth/auth.service.ts (add to existing file)

export const loginUser = async (email: string, password: string) => {
  const user = await prisma.user.findUnique({
    where: { email },
    select: {
      id: true,
      email: true,
      passwordHash: true,
      createdAt: true,
    },
  });

  // Same error for bad email or bad password — don't leak which one
  const invalid = () => new UnauthorizedError('Invalid email or password');

  if (!user) {
    // Run bcrypt anyway to prevent timing attacks
    await bcrypt.compare(password, '$2b$12$invalidhashfortimingprotection000000000000000000000');
    throw invalid();
  }

  const valid = await bcrypt.compare(password, user.passwordHash);
  if (!valid) throw invalid();

  const { passwordHash: _, ...safeUser } = user;
  const token = signToken({ userId: safeUser.id, email: safeUser.email });

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

typescript

```typescript
// src/modules/auth/auth.controller.ts (add to existing file)

async login(req: Request, res: Response, next: NextFunction) {
  try {
    const { email, password } = validateLoginInput(req.body);
    const { user, token } = await loginUser(email, password);

    res.status(200).json({ user, token });
  } catch (err) {
    next(err);
  }
},
```

typescript

```typescript
// src/modules/auth/auth.routes.ts (updated)

router.post('/register', AuthController.register);
router.post('/login', AuthController.login);
```

Example responses:

Success `200`:

json

```json
{
  "user": {
    "id": "uuid",
    "email": "jane@example.com",
    "createdAt": "2024-01-15T10:30:00.000Z"
  },
  "token": "eyJhbGci..."
}
```

Bad credentials `401`:

json

```json
{
  "error": {
    "code": "UNAUTHORIZED",
    "message": "Invalid email or password",
    "timestamp": "2024-01-15T10:30:00.000Z"
  }
}
```

Key points:

Timing attack prevention — if the user doesn't exist, we still run `bcrypt.compare` against a dummy hash. Without this, an attacker can enumerate valid emails by measuring response time (bcrypt is intentionally slow; skipping it for missing users creates a measurable difference).

Generic error message — `"Invalid email or password"` for both bad email and bad password. Never tell the client which one failed.

Destructure out `passwordHash` — `const { passwordHash: _, ...safeUser }` strips it before the response. TypeScript's _ convention signals intentional discard.

Ready to build the task CRUD endpoints?

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.

Feb 18

typescript

```typescript
// src/modules/orgs/org.validation.ts

import { ValidationError } from '../../errors';

interface CreateOrgInput {
  name: string;
  slug: string;
}
```

```
export const validateCreateOrgInput = (body: unknown): CreateOrgInput => {
  const { name, slug } = body as Record<string, unknown>;
  const errs: string[] = [];

  if (!name || typeof name !== 'string' || name.trim().length < 2) {
    errs.push('Name must be at least 2 characters');
  }

  if (!slug || typeof slug !== 'string') {
    errs.push('Slug is required');
  } else if (!/^[a-z0-9-]+$/.test(slug)) {
    errs.push('Slug may only contain lowercase letters, numbers, and hyphens');
  } else if (slug.length < 2 || slug.length > 50) {
    errs.push('Slug must be between 2 and 50 characters');
  }

  if (errs.length) throw new ValidationError('Invalid input', errs);

  return { name: name.trim(), slug: slug.toLowerCase() };
};
```

typescript

```
// src/modules/orgs/org.service.ts

import { prisma } from '../../config/database';
import { ConflictError } from '../../errors';

export const createOrganization = async (
  userId: string,
  name: string,
  slug: string
) => {
  const existing = await prisma.organization.findUnique({ where: { slug } });
  if (existing) throw new ConflictError(`Slug "${slug}" is already taken`);

  // Create org and membership in one transaction
  const org = await prisma.$transaction(async (tx) => {
    const organization = await tx.organization.create({
      data: { name, slug },
      select: { id: true, name: true, slug: true, createdAt: true },
    });

    await tx.organizationMember.create({
      data: {
        userId,
        orgId: organization.id,
        role: 'OWNER',
      },
    });

    return organization;
  });

  return org;
};
```

typescript

```
// src/modules/orgs/org.controller.ts

import { Response, NextFunction } from 'express';
import { AuthRequest } from '../../types';
import { validateCreateOrgInput } from './org.validation';
import { createOrganization } from './org.service';

export const OrgController = {
  async create(req: AuthRequest, res: Response, next: NextFunction) {
    try {
      const { name, slug } = validateCreateOrgInput(req.body);
      const org = await createOrganization(req.user!.userId, name,
slug);

      res.status(201).json({ org });
    } catch (err) {
      next(err);
    }
  },
};
```

typescript

```
// src/modules/orgs/org.routes.ts

import { Router } from 'express';
import { authenticate } from '../../middleware/auth';
import { OrgController } from './org.controller';
```

```
const router = Router();

router.post('/', authenticate, OrgController.create);

export { router as orgRoutes };
```

typescript

```
// src/app.ts (updated)

import { orgRoutes } from './modules/orgs/org.routes';

app.use('/api/auth', authRoutes);
app.use('/api/orgs', orgRoutes);
app.use('/api/orgs/:orgId/tasks', taskRoutes);
```

Example responses:

Success `201`:

json

```
{
  "org": {
    "id": "uuid",
    "name": "Acme Corp",
    "slug": "acme-corp",
    "createdAt": "2024-01-15T10:30:00.000Z"
  }
}
```

Slug taken `409`:

json

```
{
  "error": {
    "code": "CONFLICT",
    "message": "Slug \"acme-corp\" is already taken",
    "timestamp": "2024-01-15T10:30:00.000Z"
  }
}
```

Unauthenticated `401`:

json

```
{
  "error": {
    "code": "UNAUTHORIZED",
    "message": "Missing or invalid authorization header",
    "timestamp": "2024-01-15T10:30:00.000Z"
  }
}
```

Key points:

Transaction — org creation and membership are atomic. If the membership insert fails for any reason, the org is rolled back. You never end up with an org that has no owner.

OWNER not ADMIN — the creator gets `OWNER`, the highest role. Admins can be added later but can't exceed the owner's privileges.

Slug validation — lowercase + numbers + hyphens only, matching URL-safe conventions. The unique check before insert gives a clean error; the DB unique constraint is the safety net for races.

Route placement — `POST /api/orgs` only needs `authenticate`. The `resolveTenant` middleware is for routes scoped under `/:orgId` where org membership needs verifying.

Ready to build the task CRUD endpoints?

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.

Feb 18

typescript

```typescript
// src/modules/orgs/org.validation.ts (add to existing file)

export const validateAddMemberInput = (body: unknown): { email:
string; role: Role } => {
  const { email, role } = body as Record<string, unknown>;
  const errs: string[] = [];
  const validRoles: Role[] = ['ADMIN', 'MEMBER'];

  if (!email || typeof email !== 'string') {
    errs.push('Email is required');
  } else if (!/^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(email)) {
    errs.push('Invalid email format');
  }

  if (role && !validRoles.includes(role as Role)) {
    errs.push(`Role must be one of: ${validRoles.join(', ')}`);
  }

  if (errs.length) throw new ValidationError('Invalid input', errs);

  return {
    email: (email as string).toLowerCase().trim(),
    role: (role as Role) ?? 'MEMBER',
  };
};
```

typescript

```typescript
// src/modules/orgs/org.service.ts (add to existing file)

import { Role } from '@prisma/client';
import { NotFoundError, ConflictError, ForbiddenError } from '../../errors';

export const addMember = async (
  actorRole: Role,
  orgId: string,
  email: string,
  role: Role
) => {
  // Admins can only assign MEMBER — only OWNERs can assign ADMIN or higher
  if (role === 'OWNER') {
    throw new ForbiddenError('Cannot assign OWNER role');
  }

  if (actorRole === 'ADMIN' && role === 'ADMIN') {
    throw new ForbiddenError('Admins cannot assign the ADMIN role');
  }

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

  if (!targetUser) throw new NotFoundError('User');

  const existing = await prisma.organizationMember.findUnique({
    where: { userId_orgId: { userId: targetUser.id, orgId } },
  });

  if (existing) throw new ConflictError('User is already a member of this organization');

  const membership = await prisma.organizationMember.create({
    data: { userId: targetUser.id, orgId, role },
    select: {
      role: true,
      joinedAt: true,
      user: { select: { id: true, email: true } },
    },
  });

  return membership;
};
```

typescript

```
// src/modules/orgs/org.controller.ts (add to existing file)

import { validateAddMemberInput } from './org.validation';
import { addMember } from './org.service';

export const OrgController = {
  // ...existing create

  async addMember(req: AuthRequest, res: Response, next:
NextFunction) {
    try {
      const { email, role } = validateAddMemberInput(req.body);
      const { orgId, role: actorRole } = req.membership!;

      const membership = await addMember(actorRole, orgId, email,
role);

      res.status(201).json({ membership });
    } catch (err) {
      next(err);
    }
  },
};
```

typescript

```
// src/modules/orgs/org.routes.ts (updated)

import { Router } from 'express';
import { authenticate } from '../../middleware/auth';
import { resolveTenant } from '../../middleware/tenant';
import { requireRole } from '../../middleware/requireRole';
import { OrgController } from './org.controller';

const router = Router();

router.post('/', authenticate, OrgController.create);

router.post(
  '/:orgId/members',
  authenticate,
  resolveTenant,
  requireRole('ADMIN'),
  OrgController.addMember
);
```

```
export { router as orgRoutes };
```

Authorization matrix:

Actor	Assign MEMBER	Assign ADMIN	Assign OWNER
MEMBER	❌	❌	❌
ADMIN	✅	❌	❌
OWNER	✅	✅	❌

Example responses:

Success `201`:

json

```json
{
  "membership": {
    "role": "MEMBER",
    "joinedAt": "2024-01-15T10:30:00.000Z",
    "user": {
      "id": "uuid",
      "email": "jane@example.com"
    }
  }
}
```

Not an org member `403`:

json

```json
{
  "error": {
    "code": "FORBIDDEN",
    "message": "Access denied to this organization"
  }
}
```

Insufficient role `403`:

json

```
{
  "error": {
    "code": "FORBIDDEN",
    "message": "Requires ADMIN role or higher"
  }
}
```

User not found `404`:

json

```
{
  "error": {
    "code": "NOT_FOUND",
    "message": "User not found"
  }
}
```

Already a member `409`:

json

```
{
  "error": {
    "code": "CONFLICT",
    "message": "User is already a member of this organization"
  }
}
```

Key points:

Three auth layers — `authenticate` (valid JWT) → `resolveTenant` (is org member) → `requireRole('ADMIN')` (has sufficient role). All three must pass before the controller runs.

Role ceiling in service — `requireRole('ADMIN')` in the middleware lets ADMINs and OWNERs through, but the service enforces that ADMINs can't assign roles equal to or above their own. Privilege escalation is blocked at the business logic layer, not just the middleware.

OWNER is unassignable — there's always exactly one owner per org (the creator). Ownership transfer would be a separate, explicit endpoint.

Add by email not ID — callers look up users by email, which is the natural identifier. The service resolves to the internal UUID.

Ready to build the task CRUD endpoints?

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).

Feb 18

typescript

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

import { TaskStatus, Priority } from '@prisma/client';
import { ValidationError, BadRequestError } from '../../errors';

export const validateCreateTaskInput = (body: unknown) => {
  const { title, description, status, priority, assignedToId, dueDate
} =
    body as Record<string, unknown>;
  const errs: string[] = [];

  if (!title || typeof title !== 'string' || title.trim().length < 1)
{
    errs.push('Title is required');
  } else if (title.trim().length > 500) {
    errs.push('Title must be under 500 characters');
  }

  if (description && typeof description !== 'string') {
    errs.push('Description must be a string');
  }

  if (status && !Object.values(TaskStatus).includes(status as
TaskStatus)) {
    errs.push(`Status must be one of:
${Object.values(TaskStatus).join(', ')}`);
  }

  if (priority && !Object.values(Priority).includes(priority as
Priority)) {
    errs.push(`Priority must be one of:
${Object.values(Priority).join(', ')}`);
  }

  if (assignedToId && typeof assignedToId !== 'string') {
    errs.push('assignedToId must be a string');
  }

  if (dueDate && isNaN(Date.parse(dueDate as string))) {
    errs.push('dueDate must be a valid ISO date string');
  }

  if (errs.length) throw new ValidationError('Invalid input', errs);

  return {
    title: (title as string).trim(),
    description: description as string | undefined,
    status: (status as TaskStatus) ?? TaskStatus.TODO,
    priority: (priority as Priority) ?? Priority.MEDIUM,
    assignedToId: assignedToId as string | undefined,
    dueDate: dueDate ? new Date(dueDate as string) : undefined,
  };
};

export const validateUpdateTaskInput = (body: unknown) => {
  const { title, description, status, priority, assignedToId, dueDate
} =
    body as Record<string, unknown>;
  const errs: string[] = [];

  if (title !== undefined) {
    if (typeof title !== 'string' || title.trim().length < 1) {
      errs.push('Title must be a non-empty string');
    } else if (title.trim().length > 500) {
      errs.push('Title must be under 500 characters');
    }
  }

  if (status !== undefined &&
!Object.values(TaskStatus).includes(status as TaskStatus)) {
    errs.push(`Status must be one of:
${Object.values(TaskStatus).join(', ')}`);
  }

  if (priority !== undefined &&
!Object.values(Priority).includes(priority as Priority)) {
    errs.push(`Priority must be one of:
${Object.values(Priority).join(', ')}`);
  }

  if (dueDate !== undefined && dueDate !== null &&
isNaN(Date.parse(dueDate as string))) {
    errs.push('dueDate must be a valid ISO date string or null');
  }
  if (errs.length) throw new ValidationError('Invalid input', errs);

  return {
    ...(title !== undefined && { title: (title as string).trim() }),
    ...(description !== undefined && { description: description as string | null }),
    ...(status !== undefined && { status: status as TaskStatus }),
    ...(priority !== undefined && { priority: priority as Priority }),
    ...(assignedToId !== undefined && { assignedToId: assignedToId as string | null }),
    ...(dueDate !== undefined && { dueDate: dueDate ? new Date(dueDate as string) : null }),
  };
};

export const validateTaskListQuery = (query: unknown) => {
  const { status, priority, assignedToId, page, limit } =
    query as Record<string, unknown>;

  const parsedPage = page ? parseInt(page as string, 10) : 1;
  const parsedLimit = limit ? parseInt(limit as string, 10) : 20;

  if (isNaN(parsedPage) || parsedPage < 1) {
    throw new BadRequestError('Page must be a positive integer');
  }
  if (isNaN(parsedLimit) || parsedLimit < 1 || parsedLimit > 100) {
    throw new BadRequestError('Limit must be between 1 and 100');
  }

  return {
    status: status as TaskStatus | undefined,
    priority: priority as Priority | undefined,
    assignedToId: assignedToId as string | undefined,
    page: parsedPage,
    limit: parsedLimit,
  };
};
```

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

import { TaskStatus, Priority } from '@prisma/client';
import { prisma } from '../../config/database';
import { NotFoundError, ForbiddenError } from '../../errors';
```

```
const taskSelect = {
  id: true,
  title: true,
  description: true,
  status: true,
  priority: true,
  dueDate: true,
  createdAt: true,
  updatedAt: true,
  createdBy: { select: { id: true, email: true } },
  assignedTo: { select: { id: true, email: true } },
};

// Verify task exists and belongs to org — used by update/delete
const getOwnedTask = async (taskId: string, orgId: string) => {
  const task = await prisma.task.findUnique({ where: { id: taskId }
});
  if (!task) throw new NotFoundError('Task');
  if (task.orgId !== orgId) throw new ForbiddenError('Task does not
belong to this organization');
  return task;
};

export const listTasks = async (
  orgId: string,
  filters: {
    status?: TaskStatus;
    priority?: Priority;
    assignedToId?: string;
    page: number;
    limit: number;
  }
) => {
  const { status, priority, assignedToId, page, limit } = filters;
  const skip = (page - 1) * limit;

  const where = {
    orgId,
    ...(status && { status }),
    ...(priority && { priority }),
    ...(assignedToId && { assignedToId }),
  };

  const [tasks, total] = await prisma.$transaction([
    prisma.task.findMany({
      where,
      select: taskSelect,
      orderBy: { createdAt: 'desc' },
      skip,
      take: limit,
    }),
    prisma.task.count({ where }),
  ]);

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

export const getTask = async (taskId: string, orgId: string) => {
  const task = await prisma.task.findUnique({
    where: { id: taskId },
    select: { ...taskSelect, orgId: true },
  });

  if (!task) throw new NotFoundError('Task');
  if (task.orgId !== orgId) throw new ForbiddenError('Task does not
belong to this organization');

  const { orgId: _, ...safeTask } = task;
  return safeTask;
};

export const createTask = async (
  orgId: string,
  createdById: string,
  data: {
    title: string;
    description?: string;
    status: TaskStatus;
    priority: Priority;
    assignedToId?: string;
    dueDate?: Date;
  }
) => {
  // Verify assignee is an org member if provided
  if (data.assignedToId) {
    const membership = await prisma.organizationMember.findUnique({
      where: { userId_orgId: { userId: data.assignedToId, orgId } },
    });
    if (!membership) throw new ForbiddenError('Assignee is not a
member of this organization');
  }

  return prisma.task.create({
    data: { ...data, orgId, createdById },
    select: taskSelect,
  });
};

export const updateTask = async (
  taskId: string,
  orgId: string,
  data: {
    title?: string;
    description?: string | null;
    status?: TaskStatus;
    priority?: Priority;
    assignedToId?: string | null;
    dueDate?: Date | null;
  }
) => {
  await getOwnedTask(taskId, orgId);

  if (data.assignedToId) {
    const membership = await prisma.organizationMember.findUnique({
      where: { userId_orgId: { userId: data.assignedToId, orgId } },
    });
    if (!membership) throw new ForbiddenError('Assignee is not a
member of this organization');
  }

  return prisma.task.update({
    where: { id: taskId },
    data,
    select: taskSelect,
  });
};
```

```
export const deleteTask = async (taskId: string, orgId: string) => {
  await getOwnedTask(taskId, orgId);
  await prisma.task.delete({ where: { id: taskId } });
};
```

typescript

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

import { Response, NextFunction } from 'express';
import { AuthRequest } from '../../types';
import {
  validateCreateTaskInput,
  validateUpdateTaskInput,
  validateTaskListQuery,
} from './task.validation';
import {
  listTasks,
  getTask,
  createTask,
  updateTask,
  deleteTask,
} from './task.service';

export const TaskController = {
  async list(req: AuthRequest, res: Response, next: NextFunction) {
    try {
      const filters = validateTaskListQuery(req.query);
      const result = await listTasks(req.membership!.orgId, filters);
      res.json(result);
    } catch (err) {
      next(err);
    }
  },

  async get(req: AuthRequest, res: Response, next: NextFunction) {
    try {
      const task = await getTask(req.params.taskId,
req.membership!.orgId);
      res.json({ task });
    } catch (err) {
      next(err);
    }
  },
  async create(req: AuthRequest, res: Response, next: NextFunction) {
    try {
      const data = validateCreateTaskInput(req.body);
      const task = await createTask(req.membership!.orgId,
req.user!.userId, data);
      res.status(201).json({ task });
    } catch (err) {
      next(err);
    }
  },

  async update(req: AuthRequest, res: Response, next: NextFunction) {
    try {
      const data = validateUpdateTaskInput(req.body);
      const task = await updateTask(req.params.taskId,
req.membership!.orgId, data);
      res.json({ task });
    } catch (err) {
      next(err);
    }
  },

  async delete(req: AuthRequest, res: Response, next: NextFunction) {
    try {
      await deleteTask(req.params.taskId, req.membership!.orgId);
      res.status(204).send();
    } catch (err) {
      next(err);
    }
  },
};
```

typescript

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

import { Router } from 'express';
import { authenticate } from '../../middleware/auth';
import { resolveTenant } from '../../middleware/tenant';
import { requireRole } from '../../middleware/requireRole';
import { TaskController } from './task.controller';

const router = Router({ mergeParams: true });

router.use(authenticate, resolveTenant);
router.get('/', TaskController.list);
router.get('/:taskId', TaskController.get);
router.post('/', TaskController.create);
router.patch('/:taskId', TaskController.update);
router.delete('/:taskId', requireRole('ADMIN'), TaskController.delete);

export { router as taskRoutes };
```

Route map:

Method	Route	Role required
GET	`/api/orgs/:orgId/tasks`	MEMBER+
GET	`/api/orgs/:orgId/tasks/:taskId`	MEMBER+
POST	`/api/orgs/:orgId/tasks`	MEMBER+
PATCH	`/api/orgs/:orgId/tasks/:taskId`	MEMBER+
DELETE	`/api/orgs/:orgId/tasks/:taskId`	ADMIN+

Example list response:

```json
{
  "tasks": [
    {
      "id": "uuid",
      "title": "Build auth module",
      "description": "JWT-based auth",
      "status": "IN_PROGRESS",
      "priority": "HIGH",
      "dueDate": "2024-02-01T00:00:00.000Z",
      "createdAt": "2024-01-15T10:30:00.000Z",
      "updatedAt": "2024-01-15T10:30:00.000Z",
      "createdBy": { "id": "uuid", "email": "jane@example.com" },
      "assignedTo": { "id": "uuid", "email": "john@example.com" }
    }
  ],
  "pagination": {
    "total": 42,
    "page": 1,
    "limit": 20,
    "pages": 3
  }
}
```

Key points:

`getOwnedTask` helper — update and delete both call this first. It checks existence and org ownership in one place, so you can't manipulate tasks across org boundaries by guessing UUIDs.

Assignee membership check — you can only assign tasks to users who are already org members. Prevents assigning to arbitrary user IDs.

Partial updates — `validateUpdateTaskInput` uses spread conditionals so only provided fields are included in the Prisma `data` object. Prisma won't touch fields that aren't in the update payload.

Nullable fields on update — `assignedToId: null` and `dueDate: null` are valid update values to explicitly unset them, handled separately from `undefined` (not provided).

`taskSelect` constant — one place defines what gets returned from every query. `passwordHash` and internal fields never leak, and you update the shape in one spot.

Pagination — `$transaction([findMany, count])` runs both queries atomically so the total count matches the result set.

Want to add filtering, sorting, or move on to testing?

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

Feb 18

Pagination is already fully built into the task list endpoint we just created. Here's a summary of what's already there:

Validation (`validateTaskListQuery`):

typescript

```typescript
page: page ? parseInt(page, 10) : 1,
limit: limit ? parseInt(limit, 10) : 20,  // capped at 100
```

Service (`listTasks`):

typescript

```typescript
const skip = (page - 1) * limit;

const [tasks, total] = await prisma.$transaction([
```

~~~~
  prisma.task.findMany({ where, skip, take: limit, orderBy: {
createdAt: 'desc' } }),
  prisma.task.count({ where }),
]);

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

Response shape:

json

~~~~
{
  "tasks": [...],
  "pagination": {
    "total": 42,
    "page": 2,
    "limit": 20,
    "pages": 3
  }
}
```

**Usage:**
```
GET /api/orgs/:orgId/tasks?page=2&limit=20
GET /api/orgs/:orgId/tasks?page=1&limit=50&status=TODO
~~~~

Do you want cursor-based pagination instead (better for large datasets / infinite scroll), or want to move on to something new?

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.

Feb 18

Status and assignee filtering are already in place. We just need to add date range. Here's the delta:

typescript

~~~~
// src/modules/tasks/task.validation.ts — replace
validateTaskListQuery
~~~~

```
export const validateTaskListQuery = (query: unknown) => {
  const { status, priority, assignedToId, page, limit, dueDateFrom,
dueDateTo } =
    query as Record<string, unknown>;

  const parsedPage = page ? parseInt(page as string, 10) : 1;
  const parsedLimit = limit ? parseInt(limit as string, 10) : 20;
  const errs: string[] = [];

  if (isNaN(parsedPage) || parsedPage < 1) errs.push('Page must be a
positive integer');
  if (isNaN(parsedLimit) || parsedLimit < 1 || parsedLimit > 100)
errs.push('Limit must be between 1 and 100');
  if (dueDateFrom && isNaN(Date.parse(dueDateFrom as string)))
errs.push('dueDateFrom must be a valid ISO date');
  if (dueDateTo && isNaN(Date.parse(dueDateTo as string)))
errs.push('dueDateTo must be a valid ISO date');
  if (dueDateFrom && dueDateTo && new Date(dueDateFrom as string) >
new Date(dueDateTo as string)) {
    errs.push('dueDateFrom must be before dueDateTo');
  }

  if (errs.length) throw new ValidationError('Invalid input', errs);

  return {
    status: status as TaskStatus | undefined,
    priority: priority as Priority | undefined,
    assignedToId: assignedToId as string | undefined,
    dueDateFrom: dueDateFrom ? new Date(dueDateFrom as string) :
undefined,
    dueDateTo: dueDateTo ? new Date(dueDateTo as string) : undefined,
    page: parsedPage,
    limit: parsedLimit,
  };
};
```

typescript

```
// src/modules/tasks/task.service.ts — replace listTasks

export const listTasks = async (
  orgId: string,
  filters: {
    status?: TaskStatus;
    priority?: Priority;
    assignedToId?: string;
    dueDateFrom?: Date;
    dueDateTo?: Date;
    page: number;
    limit: number;
  }
) => {
  const { status, priority, assignedToId, dueDateFrom, dueDateTo,
page, limit } = filters;
  const skip = (page - 1) * limit;

  const where = {
    orgId,
    ...(status && { status }),
    ...(priority && { priority }),
    ...(assignedToId && { assignedToId }),
    ...(dueDateFrom || dueDateTo
      ? {
          dueDate: {
            ...(dueDateFrom && { gte: dueDateFrom }),
            ...(dueDateTo && { lte: dueDateTo }),
          },
        }
      : {}),
  };

  const [tasks, total] = await prisma.$transaction([
    prisma.task.findMany({
      where,
      select: taskSelect,
      orderBy: { createdAt: 'desc' },
      skip,
      take: limit,
    }),
    prisma.task.count({ where }),
  ]);

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

~~~~
};
```

**Usage:**
```
# Filter by status
GET /api/orgs/:orgId/tasks?status=TODO

# Filter by assignee
GET /api/orgs/:orgId/tasks?assignedToId=uuid

# Date range
GET /api/orgs/:orgId/tasks?dueDateFrom=2024-01-01&dueDateTo=2024-01-31

# All combined with pagination
GET /api/orgs/:orgId/tasks?status=IN_PROGRESS&assignedToId=uuid&dueDateFrom=2024-01-01&page=2&limit=10
~~~~

That's it — the `where` object composes cleanly so all filters and pagination work together. The `count` query uses the same `where` so pagination metadata always reflects the filtered result set.

Want to add sorting next, or move on to something else?

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.

Feb 18

1. Update the Prisma schema enum:

prisma

```
// prisma/schema.prisma — replace Role enum

enum Role {
  OWNER
  ADMIN
  MEMBER
  VIEWER
}
```

bash

```
npx prisma migrate dev --name add_viewer_role
```

---

2. Update the role hierarchy:

typescript

```
// src/middleware/requireRole.ts — replace roleHierarchy

const roleHierarchy: Record<Role, number> = {
  OWNER: 4,
  ADMIN: 3,
  MEMBER: 2,
  VIEWER: 1,
};
```

---

3. Replace `requireRole` with a permissions map:

typescript

```
// src/middleware/permissions.ts

import { Role } from '@prisma/client';
import { Response, NextFunction } from 'express';
import { AuthRequest } from '../types';
import { ForbiddenError } from '../errors';

const roleHierarchy: Record<Role, number> = {
  OWNER: 4,
  ADMIN: 3,
  MEMBER: 2,
  VIEWER: 1,
};

export type Action =
  | 'task:read'
  | 'task:create'
  | 'task:update'
  | 'task:delete'
  | 'member:add'
  | 'member:remove'
  | 'member:list'
  | 'org:update'
  | 'org:delete';

const permissions: Record<Action, Role> = {
  'task:read':     'VIEWER',    // all members
  'task:create':   'MEMBER',
  'task:update':   'MEMBER',
  'task:delete':  'ADMIN',
  'member:add':   'ADMIN',
  'member:remove':'ADMIN',
  'member:list':  'VIEWER',
  'org:update':   'ADMIN',
  'org:delete':   'OWNER',
};

export const can = (role: Role, action: Action): boolean => {
  const required = permissions[action];
  return roleHierarchy[role] >= roleHierarchy[required];
};

export const requirePermission = (action: Action) => {
  return (req: AuthRequest, res: Response, next: NextFunction) => {
    const role = req.membership?.role;
    if (!role || !can(role, action)) {
      return next(new ForbiddenError(`Missing permission:
${action}`));
    }
    next();
  };
};
```

---

4. Update task routes:

typescript

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

import { Router } from 'express';
import { authenticate } from '../../middleware/auth';
import { resolveTenant } from '../../middleware/tenant';
import { requirePermission } from '../../middleware/permissions';
import { TaskController } from './task.controller';

const router = Router({ mergeParams: true });

router.use(authenticate, resolveTenant);

router.get('/',          requirePermission('task:read'),
TaskController.list);
router.get('/:taskId',   requirePermission('task:read'),
TaskController.get);
router.post('/',         requirePermission('task:create'),
TaskController.create);
```

```
router.patch('/:taskId', requirePermission('task:update'),
TaskController.update);
router.delete('/:taskId',requirePermission('task:delete'),
TaskController.delete);

export { router as taskRoutes };
```

---

5. Update org routes:

typescript

```
// src/modules/orgs/org.routes.ts

import { Router } from 'express';
import { authenticate } from '../../middleware/auth';
import { resolveTenant } from '../../middleware/tenant';
import { requirePermission } from '../../middleware/permissions';
import { OrgController } from './org.controller';

const router = Router();

router.post('/', authenticate, OrgController.create);

router.get(
  '/:orgId/members',
  authenticate, resolveTenant,
  requirePermission('member:list'),
  OrgController.listMembers
);

router.post(
  '/:orgId/members',
  authenticate, resolveTenant,
  requirePermission('member:add'),
  OrgController.addMember
);

router.delete(
  '/:orgId/members/:userId',
  authenticate, resolveTenant,
  requirePermission('member:remove'),
  OrgController.removeMember
);

export { router as orgRoutes };
```

---

6. Update `addMember` service to respect the new role:

typescript

```
// src/modules/orgs/org.service.ts — replace addMember role check

export const addMember = async (actorRole: Role, orgId: string,
email: string, role: Role) => {
  if (role === 'OWNER') {
    throw new ForbiddenError('Cannot assign OWNER role');
  }

  // Actors can only assign roles strictly below their own
  if (roleHierarchy[actorRole] <= roleHierarchy[role]) {
    throw new ForbiddenError(`A ${actorRole} cannot assign the
${role} role`);
  }

  const targetUser = await prisma.user.findUnique({ where: { email },
select: { id: true, email: true } });
  if (!targetUser) throw new NotFoundError('User');

  const existing = await prisma.organizationMember.findUnique({
    where: { userId_orgId: { userId: targetUser.id, orgId } },
  });
  if (existing) throw new ConflictError('User is already a member of
this organization');

  return prisma.organizationMember.create({
    data: { userId: targetUser.id, orgId, role },
    select: { role: true, joinedAt: true, user: { select: { id: true,
email: true } } },
  });
};

const roleHierarchy: Record<Role, number> = {
  OWNER: 4, ADMIN: 3, MEMBER: 2, VIEWER: 1,
};
```

Permission matrix:

Action	VIEWER	MEMBER	ADMIN	OWNER

task:read	✅	✅	✅	✅
task:create	❌	✅	✅	✅
task:update	❌	✅	✅	✅
task:delete	❌	❌	✅	✅
member:list	✅	✅	✅	✅
member:add	❌	❌	✅	✅
member:remove	❌	❌	✅	✅
org:update	❌	❌	✅	✅
org:delete	❌	❌	❌	✅

Error response for insufficient permission `403`:

```json
{
  "error": {
    "code": "FORBIDDEN",
    "message": "Missing permission: task:create",
    "timestamp": "2024-01-15T10:30:00.000Z"
  }
}
```

Key points:

`permissions` map is the single source of truth — adding a new action is one line. No scattered role checks across controllers.

`can()` is exported — useful for conditional logic inside services, e.g. members can only edit their own tasks but admins can edit any. Call `can(role, 'task:update')` directly without going through middleware.

Role assignment ceiling — actors can only assign roles strictly below their own. An ADMIN can add MEMBERs and VIEWERs but not other ADMINs.

One migration — just adding `VIEWER` to the enum. Existing `MEMBER` rows are unaffected.

Ready to add tests, or what's next?

Refactor the task endpoints to use the new permission system. Admins and Members can create/edit, but Viewers can only read.

Feb 18

Already done. The task routes are already using `requirePermission` from the system we just built:

typescript

```
router.get('/',          requirePermission('task:read'),   TaskController.list);
router.get('/:taskId',   requirePermission('task:read'),   TaskController.get);
router.post('/',         requirePermission('task:create'), TaskController.create);
router.patch('/:taskId', requirePermission('task:update'), TaskController.update);
router.delete('/:taskId',requirePermission('task:delete'), TaskController.delete);
```

And the permissions map already enforces exactly what you described:

typescript

```
'task:read':   'VIEWER',   // VIEWER, MEMBER, ADMIN, OWNER
'task:create': 'MEMBER',   // MEMBER, ADMIN, OWNER — not VIEWER
'task:update': 'MEMBER',   // MEMBER, ADMIN, OWNER — not VIEWER
'task:delete': 'ADMIN',    // ADMIN, OWNER only
```

Nothing to change. Want to add something on top of this — like members can only edit their own tasks — or move on to something new?

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.

Feb 18

Already covered. Here's what's already validating each of those:

Email format — `auth.validation.ts`:

typescript

```
if (!/^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(email))
```

```
  errs.push('Invalid email format');
```

Required fields for task creation — `task.validation.ts`:

typescript

```typescript
if (!title || typeof title !== 'string' || title.trim().length < 1)
  errs.push('Title is required');
```

Enum values — `task.validation.ts`:

typescript

```typescript
if (status && !Object.values(TaskStatus).includes(status as
TaskStatus))
  errs.push(`Status must be one of:
${Object.values(TaskStatus).join(', ')}`);

if (priority && !Object.values(Priority).includes(priority as
Priority))
  errs.push(`Priority must be one of:
${Object.values(Priority).join(', ')}`);
```

Role enum — `org.validation.ts`:

typescript

```typescript
const validRoles: Role[] = ['ADMIN', 'MEMBER'];
if (role && !validRoles.includes(role as Role))
  errs.push(`Role must be one of: ${validRoles.join(', ')}`);
```

All validation throws `ValidationError` with collected errors, returning `422` with a `details` array.

That said, if you want to switch to Zod for more robust, schema-first validation, here's the migration. It's a meaningful upgrade — better type inference, less boilerplate, and schemas double as TypeScript types.

bash

```bash
npm install zod
```

typescript

```typescript
// src/lib/validate.ts

import { z, ZodSchema } from 'zod';
import { ValidationError } from '../errors';

export const validate = <T>(schema: ZodSchema<T>, data: unknown): T
=> {
  const result = schema.safeParse(data);
  if (!result.success) {
    const errors = result.error.errors.map((e) =>
`${e.path.join('.')}: ${e.message}`);
    throw new ValidationError('Invalid input', errors);
  }
  return result.data;
};
```

typescript

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

import { z } from 'zod';
import { validate } from '../../lib/validate';

const registerSchema = z.object({
  email: z.string().email('Invalid email format').transform((v) =>
v.toLowerCase().trim()),
  password: z
    .string()
    .min(8, 'Password must be at least 8 characters')
    .regex(/(?=.*[a-z])(?=.*[A-Z])(?=.*\d)/, 'Password must contain
uppercase, lowercase, and a number'),
});

const loginSchema = z.object({
  email: z.string().email('Invalid email format').transform((v) =>
v.toLowerCase().trim()),
  password: z.string().min(1, 'Password is required'),
});

export type RegisterInput = z.infer<typeof registerSchema>;
export type LoginInput = z.infer<typeof loginSchema>;

export const validateRegisterInput = (body: unknown): RegisterInput
=>
  validate(registerSchema, body);

export const validateLoginInput = (body: unknown): LoginInput =>
  validate(loginSchema, body);
```

typescript

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

import { z } from 'zod';
import { TaskStatus, Priority } from '@prisma/client';
import { validate } from '../../lib/validate';

const taskStatusEnum = z.nativeEnum(TaskStatus);
const priorityEnum = z.nativeEnum(Priority);

const createTaskSchema = z.object({
  title: z.string().min(1, 'Title is required').max(500, 'Title must
be under 500 characters').transform((v) => v.trim()),
  description: z.string().optional(),
  status: taskStatusEnum.default(TaskStatus.TODO),
  priority: priorityEnum.default(Priority.MEDIUM),
  assignedToId: z.string().uuid('assignedToId must be a valid
UUID').optional(),
  dueDate: z.string().datetime('dueDate must be a valid ISO
date').transform((v) => new Date(v)).optional(),
});

const updateTaskSchema = z.object({
  title: z.string().min(1).max(500).transform((v) =>
v.trim()).optional(),
  description: z.string().nullable().optional(),
  status: taskStatusEnum.optional(),
  priority: priorityEnum.optional(),
  assignedToId: z.string().uuid().nullable().optional(),
  dueDate: z.string().datetime().transform((v) => new
Date(v)).nullable().optional(),
});

const taskListQuerySchema = z.object({
  status: taskStatusEnum.optional(),
  priority: priorityEnum.optional(),
  assignedToId: z.string().uuid().optional(),
  dueDateFrom: z.string().datetime().transform((v) => new
Date(v)).optional(),
  dueDateTo: z.string().datetime().transform((v) => new
Date(v)).optional(),
  page: z.coerce.number().int().min(1).default(1),
  limit: z.coerce.number().int().min(1).max(100).default(20),
}).refine(
  (data) => !data.dueDateFrom || !data.dueDateTo || data.dueDateFrom
<= data.dueDateTo,
  { message: 'dueDateFrom must be before dueDateTo', path:
['dueDateFrom'] }
);
```

```typescript
export type CreateTaskInput = z.infer<typeof createTaskSchema>;
export type UpdateTaskInput = z.infer<typeof updateTaskSchema>;
export type TaskListQuery = z.infer<typeof taskListQuerySchema>;

export const validateCreateTaskInput = (body: unknown): CreateTaskInput =>
  validate(createTaskSchema, body);

export const validateUpdateTaskInput = (body: unknown): UpdateTaskInput =>
  validate(updateTaskSchema, body);

export const validateTaskListQuery = (query: unknown): TaskListQuery =>
  validate(taskListQuerySchema, query);
```

typescript

```typescript
// src/modules/orgs/org.validation.ts

import { z } from 'zod';
import { Role } from '@prisma/client';
import { validate } from '../../lib/validate';

const createOrgSchema = z.object({
  name: z.string().min(2, 'Name must be at least 2 characters').transform((v) => v.trim()),
  slug: z
    .string()
    .min(2)
    .max(50)
    .regex(/^[a-z0-9-]+$/, 'Slug may only contain lowercase letters, numbers, and hyphens')
    .transform((v) => v.toLowerCase()),
});

const addMemberSchema = z.object({
  email: z.string().email('Invalid email format').transform((v) => v.toLowerCase().trim()),
  role: z.enum(['ADMIN', 'MEMBER', 'VIEWER'] as const).default('MEMBER'),
});

export type CreateOrgInput = z.infer<typeof createOrgSchema>;
export type AddMemberInput = z.infer<typeof addMemberSchema>;
export const validateCreateOrgInput = (body: unknown): CreateOrgInput
=>
  validate(createOrgSchema, body);

export const validateAddMemberInput = (body: unknown): AddMemberInput
=>
  validate(addMemberSchema, body);
```

Validation error response `422`:

json

```
{
  "error": {
    "code": "VALIDATION_ERROR",
    "message": "Invalid input",
    "details": [
      "email: Invalid email format",
      "password: Password must contain uppercase, lowercase, and a
number"
    ],
    "timestamp": "2024-01-15T10:30:00.000Z"
  }
}
```

Key points:

`validate()` wrapper — one function bridges Zod and your error system. All schemas throw the same `ValidationError` shape regardless of which module they're in.

`z.nativeEnum` — uses your Prisma-generated enums directly, so valid values stay in sync with the DB automatically.

`z.infer` — schemas generate TypeScript types, eliminating separate interface definitions. The schema is the type.

`.refine()` — cross-field validation like date range ordering that can't be expressed per-field.

`z.coerce.number()` — query string params arrive as strings; coerce handles the cast cleanly without manual `parseInt`.

Ready to write tests, or what's next?

Create a middleware to ensure users can only access organizations they belong to. This should be reusable across all organization-scoped endpoints.

Feb 18

Already done. `resolveTenant` in `src/middleware/tenant.ts` is exactly that middleware:

typescript

```
export const resolveTenant = async (req, res, next) => {
  const membership = await prisma.organizationMember.findUnique({
    where: { userId_orgId: { userId: req.user!.userId, orgId: 
req.params.orgId } },
  });

  if (!membership) return next(new ForbiddenError('Access denied to 
this organization'));

  req.membership = { orgId, role: membership.role };
  next();
};
```

And it's already applied to every org-scoped route:

typescript

```
// tasks
router.use(authenticate, resolveTenant);

// org members
router.post('/:orgId/members', authenticate, resolveTenant, 
requirePermission('member:add'), ...);
router.get('/:orgId/members',  authenticate, resolveTenant, 
requirePermission('member:list'), ...);
```

Any route that uses `resolveTenant` will 403 if the user isn't a member. `req.membership` is then available downstream for role/permission checks.

The only thing worth adding if your API grows is a check for suspended/inactive orgs. Want that, or ready to move on?

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.

Feb 18

1. Schema update:

prisma

```
// prisma/schema.prisma — add model

model AuditLog {
  id         String   @id @default(uuid())
  orgId      String   @map("org_id")
  userId     String   @map("user_id")
  action     AuditAction
  entityType String   @map("entity_type")
  entityId   String   @map("entity_id")
  before     Json?
  after      Json?
  createdAt  DateTime @default(now()) @map("created_at")

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

  @@index([orgId])
  @@index([userId])
  @@index([entityId])
  @@index([orgId, entityType, action])
  @@map("audit_logs")
}

enum AuditAction {
  CREATED
  UPDATED
  DELETED
}
```

Add `auditLogs AuditLog[]` relation to both `Organization` and `User` models, then:

bash

```bash
npx prisma migrate dev --name add_audit_logs
```

---

2. Audit service:

typescript

```typescript
// src/modules/audit/audit.service.ts

import { AuditAction, Prisma } from '@prisma/client';
import { prisma } from '../../config/database';

interface LogAuditParams {
  orgId: string;
  userId: string;
  action: AuditAction;
  entityType: string;
  entityId: string;
  before?: Record<string, unknown>;
  after?: Record<string, unknown>;
  tx?: Prisma.TransactionClient;
}
```

```
export const logAudit = async ({
  orgId,
  userId,
  action,
  entityType,
  entityId,
  before,
  after,
  tx,
}: LogAuditParams) => {
  const client = tx ?? prisma;

  await client.auditLog.create({
    data: { orgId, userId, action, entityType, entityId, before, after },
  });
};

export const diffObjects = (
  before: Record<string, unknown>,
  after: Record<string, unknown>
): { before: Record<string, unknown>; after: Record<string, unknown> } => {
  const changedKeys = Object.keys(after).filter(
    (key) => JSON.stringify(before[key]) !== JSON.stringify(after[key])
  );

  return {
    before: Object.fromEntries(changedKeys.map((k) => [k, before[k]])),
    after: Object.fromEntries(changedKeys.map((k) => [k, after[k]])),
  };
};
```

---

3. Update task service:

typescript

```typescript
// src/modules/tasks/task.service.ts — replace createTask, updateTask, deleteTask

export const createTask = async (
  orgId: string,
  createdById: string,
  data: CreateTaskInput
) => {
  if (data.assignedToId) {
    const membership = await prisma.organizationMember.findUnique({
      where: { userId_orgId: { userId: data.assignedToId, orgId } },
    });
    if (!membership) throw new ForbiddenError('Assignee is not a
member of this organization');
  }

  return prisma.$transaction(async (tx) => {
    const task = await tx.task.create({
      data: { ...data, orgId, createdById },
      select: taskSelect,
    });

    await logAudit({
      orgId,
      userId: createdById,
      action: 'CREATED',
      entityType: 'task',
      entityId: task.id,
      after: task as Record<string, unknown>,
      tx,
    });

    return task;
  });
};

export const updateTask = async (
  taskId: string,
  orgId: string,
  userId: string,
  data: UpdateTaskInput
) => {
  const existing = await getOwnedTask(taskId, orgId);

  if (data.assignedToId) {
    const membership = await prisma.organizationMember.findUnique({
      where: { userId_orgId: { userId: data.assignedToId, orgId } },
    });
    if (!membership) throw new ForbiddenError('Assignee is not a
member of this organization');
  }
  return prisma.$transaction(async (tx) => {
    const task = await tx.task.update({
      where: { id: taskId },
      data,
      select: taskSelect,
    });

    const { before, after } = diffObjects(
      existing as unknown as Record<string, unknown>,
      task as unknown as Record<string, unknown>
    );

    await logAudit({
      orgId,
      userId,
      action: 'UPDATED',
      entityType: 'task',
      entityId: taskId,
      before,
      after,
      tx,
    });

    return task;
  });
};

export const deleteTask = async (taskId: string, orgId: string,
userId: string) => {
  const existing = await getOwnedTask(taskId, orgId);

  await prisma.$transaction(async (tx) => {
    await tx.task.delete({ where: { id: taskId } });

    await logAudit({
      orgId,
      userId,
      action: 'DELETED',
      entityType: 'task',
      entityId: taskId,
      before: existing as unknown as Record<string, unknown>,
      tx,
    });
  });
```

```
};
```

---

4. Pass `userId` from controllers:

typescript

```
// src/modules/tasks/task.controller.ts — update update + delete

async update(req: AuthRequest, res: Response, next: NextFunction) {
  try {
    const data = validateUpdateTaskInput(req.body);
    const task = await updateTask(req.params.taskId,
req.membership!.orgId, req.user!.userId, data);
    res.json({ task });
  } catch (err) { next(err); }
},

async delete(req: AuthRequest, res: Response, next: NextFunction) {
  try {
    await deleteTask(req.params.taskId, req.membership!.orgId,
req.user!.userId);
    res.status(204).send();
  } catch (err) { next(err); }
},
```

---

5. Audit log query endpoint:

typescript

```
// src/modules/audit/audit.controller.ts

import { Response, NextFunction } from 'express';
import { AuthRequest } from '../../types';
import { prisma } from '../../config/database';
import { BadRequestError } from '../../errors';

export const AuditController = {
  async list(req: AuthRequest, res: Response, next: NextFunction) {
    try {
      const { entityType, entityId, userId, page = '1', limit = '20'
} =
        req.query as Record<string, string>;

      const parsedPage = parseInt(page, 10);
      const parsedLimit = Math.min(parseInt(limit, 10), 100);

      if (isNaN(parsedPage) || isNaN(parsedLimit)) {
        throw new BadRequestError('Invalid pagination params');
      }

      const where = {
        orgId: req.membership!.orgId,
        ...(entityType && { entityType }),
        ...(entityId && { entityId }),
        ...(userId && { userId }),
      };

      const [logs, total] = await prisma.$transaction([
        prisma.auditLog.findMany({
          where,
          orderBy: { createdAt: 'desc' },
          skip: (parsedPage - 1) * parsedLimit,
          take: parsedLimit,
          select: {
            id: true,
            action: true,
            entityType: true,
            entityId: true,
            before: true,
            after: true,
            createdAt: true,
            user: { select: { id: true, email: true } },
          },
        }),
        prisma.auditLog.count({ where }),
      ]);

      res.json({
        logs,
        pagination: {
          total,
          page: parsedPage,
          limit: parsedLimit,
          pages: Math.ceil(total / parsedLimit),
        },
      });
    } catch (err) {
      next(err);
    }
  },
};
```

```typescript
// src/modules/audit/audit.routes.ts

import { Router } from 'express';
import { authenticate } from '../../middleware/auth';
import { resolveTenant } from '../../middleware/tenant';
import { requirePermission } from '../../middleware/permissions';
import { AuditController } from './audit.controller';

const router = Router({ mergeParams: true });

router.use(authenticate, resolveTenant);

router.get('/', requirePermission('task:read'),
AuditController.list);

export { router as auditRoutes };
```

```typescript
// src/app.ts — add route

app.use('/api/orgs/:orgId/audit', auditRoutes);
```

Example audit log response:

```json
{
  "logs": [
    {
      "id": "uuid",
      "action": "UPDATED",
      "entityType": "task",
      "entityId": "uuid",
      "before": { "status": "TODO", "assignedToId": null },
      "after":  { "status": "IN_PROGRESS", "assignedToId": "uuid" },
      "createdAt": "2024-01-15T10:30:00.000Z",
      "user": { "id": "uuid", "email": "jane@example.com" }
    }
  ],
  "pagination": { "total": 1, "page": 1, "limit": 20, "pages": 1 }
}
```

Key points:

`diffObjects` — stores only changed fields in `before`/`after`, not the full snapshot. Keeps audit rows lean and makes it immediately obvious what changed.

Same transaction — the audit write is in the same `$transaction` as the task mutation. If the task write fails, no audit row is created. If the audit write fails, the task change rolls back. They're always in sync.

`tx` param on `logAudit` — falls back to the global `prisma` client when called outside a transaction, keeping it usable anywhere.

`before` on delete — stores the full task snapshot so you have a complete record of what was deleted and can reconstruct it if needed.

Ready to add tests or what's next?

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.

Feb 18

1. Schema update:

prisma

```
// prisma/schema.prisma — add to Task model

model Task {
  // ...existing fields
  deletedAt   DateTime? @map("deleted_at")
  deletedById String?   @map("deleted_by")

  deletedBy User? @relation("TaskDeleter", fields: [deletedById],
references: [id], onDelete: SetNull)
}
```

Add to `User` model:

prisma

```
deletedTasks Task[] @relation("TaskDeleter")
```

bash

```
npx prisma migrate dev --name task_soft_delete
```

---

2. Update `taskSelect` and `getOwnedTask`:

typescript

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

const taskSelect = {
  id: true,
  title: true,
  description: true,
  status: true,
  priority: true,
  dueDate: true,
  deletedAt: true,
  createdAt: true,
  updatedAt: true,
  createdBy:  { select: { id: true, email: true } },
  assignedTo: { select: { id: true, email: true } },
};

// Exclude soft-deleted tasks from ownership check
const getOwnedTask = async (taskId: string, orgId: string) => {
  const task = await prisma.task.findUnique({ where: { id: taskId }
});
  if (!task || task.deletedAt) throw new NotFoundError('Task');
  if (task.orgId !== orgId) throw new ForbiddenError('Task does not
belong to this organization');
  return task;
};
```

---

3. Update all task queries to exclude soft-deleted:

typescript

```
// listTasks — add deletedAt filter to where
const where = {
  orgId,
  deletedAt: null,                 // exclude soft-deleted
  ...(status && { status }),
  ...(priority && { priority }),
  ...(assignedToId && { assignedToId }),
  ...(dueDateFrom || dueDateTo ? {
    dueDate: {
      ...(dueDateFrom && { gte: dueDateFrom }),
      ...(dueDateTo && { lte: dueDateTo }),
    },
  } : {}),
};

// getTask — add deletedAt check
export const getTask = async (taskId: string, orgId: string) => {
  const task = await prisma.task.findUnique({
    where: { id: taskId },
    select: { ...taskSelect, orgId: true },
  });

  if (!task || task.deletedAt) throw new NotFoundError('Task');
  if (task.orgId !== orgId) throw new ForbiddenError('Task does not
belong to this organization');

  const { orgId: _, ...safeTask } = task;
  return safeTask;
};
```

---

4. Replace hard delete with soft delete:

typescript

```
// src/modules/tasks/task.service.ts — replace deleteTask

export const deleteTask = async (taskId: string, orgId: string,
userId: string) => {
  const existing = await getOwnedTask(taskId, orgId);

  await prisma.$transaction(async (tx) => {
    await tx.task.update({
      where: { id: taskId },
      data: {
        deletedAt: new Date(),
        deletedById: userId,
      },
    });

    await logAudit({
      orgId,
      userId,
      action: 'DELETED',
      entityType: 'task',
      entityId: taskId,
      before: existing as unknown as Record<string, unknown>,
      tx,
    });
  });
};
```

---

5. Add restore and list-deleted endpoints:

typescript

```
// src/modules/tasks/task.service.ts — add

export const restoreTask = async (taskId: string, orgId: string,
userId: string) => {
  const task = await prisma.task.findUnique({ where: { id: taskId }
});

  if (!task) throw new NotFoundError('Task');
  if (task.orgId !== orgId) throw new ForbiddenError('Task does not
belong to this organization');
  if (!task.deletedAt) throw new BadRequestError('Task is not
deleted');

  return prisma.$transaction(async (tx) => {
    const restored = await tx.task.update({
      where: { id: taskId },
      data: { deletedAt: null, deletedById: null },
      select: taskSelect,
    });

    await logAudit({
      orgId,
      userId,
      action: 'UPDATED',
      entityType: 'task',
      entityId: taskId,
      before: { deletedAt: task.deletedAt },
      after:  { deletedAt: null },
      tx,
    });

    return restored;
  });
};

export const listDeletedTasks = async (orgId: string, page: number,
limit: number) => {
  const where = { orgId, deletedAt: { not: null } };

  const [tasks, total] = await prisma.$transaction([
    prisma.task.findMany({
      where,
      select: {
        ...taskSelect,
        deletedBy: { select: { id: true, email: true } },
      },
      orderBy: { deletedAt: 'desc' },
      skip: (page - 1) * limit,
      take: limit,
    }),
    prisma.task.count({ where }),
  ]);

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

typescript

```
// src/modules/tasks/task.controller.ts — add

async restore(req: AuthRequest, res: Response, next: NextFunction) {
  try {
    const task = await restoreTask(req.params.taskId, req.membership!.orgId, req.user!.userId);
    res.json({ task });
  } catch (err) { next(err); }
},

async listDeleted(req: AuthRequest, res: Response, next: NextFunction) {
  try {
    const page = parseInt(req.query.page as string || '1', 10);
    const limit = Math.min(parseInt(req.query.limit as string || '20', 10), 100);
    const result = await listDeletedTasks(req.membership!.orgId, page, limit);
    res.json(result);
  } catch (err) { next(err); }
},
```

typescript

```
// src/modules/tasks/task.routes.ts — add routes

router.get('/deleted',            requirePermission('task:read'), TaskController.listDeleted);
router.delete('/:taskId',         requirePermission('task:delete'), TaskController.delete);
router.post('/:taskId/restore',   requirePermission('task:delete'), TaskController.restore);
```

Route map update:

Method	Route	Description
GET	`/tasks`	Active tasks only
DELETE	`/tasks/:id`	Soft delete
GET	`/tasks/deleted`	Deleted tasks
POST	`/tasks/:id/restore`	Restore deleted task

Deleted task response:

json

```
{
  "id": "uuid",
  "title": "Build auth module",
  "status": "TODO",
  "deletedAt": "2024-01-15T10:30:00.000Z",
  "deletedBy": { "id": "uuid", "email": "jane@example.com" }
}
```

Key points:

`deletedAt: null` in every active query — `listTasks` and `getTask` both filter this out. Soft-deleted tasks are invisible to normal operations without any extra thought.

`getOwnedTask` checks `deletedAt` — update on a deleted task returns 404, same as a nonexistent task. No special-casing needed in controllers.

Restore requires `task:delete` permission — same role that can delete can restore. Keeps the permission surface small.

`/tasks/deleted` route order — declared before `/:taskId` so Express doesn't interpret `deleted` as a task ID.

Ready to write tests or what's next?

Add task comments. Users should be able to comment on tasks. Comments belong to tasks and have:
- Content
- Author
- Timestamp

Create the schema and endpoints (create comment, list comments for a task).

Feb 18

1. Schema:

prisma

```prisma
// prisma/schema.prisma — add model

model Comment {
  id        String   @id @default(uuid())
  taskId    String   @map("task_id")
  authorId  String   @map("author_id")
  content   String
  createdAt DateTime @default(now()) @map("created_at")
  updatedAt DateTime @updatedAt @map("updated_at")

  task   Task @relation(fields: [taskId], references: [id], onDelete:
Cascade)
  author User @relation(fields: [authorId], references: [id],
onDelete: Cascade)

  @@index([taskId])
  @@map("comments")
}
```

Add to `Task` model: `comments Comment[]` Add to `User` model: `comments Comment[]`

bash

```bash
npx prisma migrate dev --name add_comments
```

---

2. Validation:

typescript

```typescript
// src/modules/comments/comment.validation.ts

import { z } from 'zod';
import { validate } from '../../lib/validate';

const createCommentSchema = z.object({
  content: z.string().min(1, 'Content is required').max(5000,
'Content must be under 5000 characters').transform((v) => v.trim()),
});

export type CreateCommentInput = z.infer<typeof createCommentSchema>;
export const validateCreateCommentInput = (body: unknown):
CreateCommentInput =>
  validate(createCommentSchema, body);
```

---

3. Service:

typescript

```typescript
// src/modules/comments/comment.service.ts

import { prisma } from '../../config/database';
import { NotFoundError, ForbiddenError } from '../../errors';
import { logAudit } from '../audit/audit.service';

const commentSelect = {
  id: true,
  content: true,
  createdAt: true,
  updatedAt: true,
  author: { select: { id: true, email: true } },
};

// Verify task exists, belongs to org, and is not soft-deleted
const resolveTask = async (taskId: string, orgId: string) => {
  const task = await prisma.task.findUnique({ where: { id: taskId } });
  if (!task || task.deletedAt) throw new NotFoundError('Task');
  if (task.orgId !== orgId) throw new ForbiddenError('Task does not belong to this organization');
  return task;
};

export const listComments = async (
  taskId: string,
  orgId: string,
  page: number,
  limit: number
) => {
  await resolveTask(taskId, orgId);

  const where = { taskId };

  const [comments, total] = await prisma.$transaction([
    prisma.comment.findMany({
      where,
      select: commentSelect,
      orderBy: { createdAt: 'asc' },
      skip: (page - 1) * limit,
      take: limit,
    }),
    prisma.comment.count({ where }),
  ]);

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

export const createComment = async (
  taskId: string,
  orgId: string,
  authorId: string,
  content: string
) => {
  await resolveTask(taskId, orgId);

  return prisma.$transaction(async (tx) => {
    const comment = await tx.comment.create({
      data: { taskId, authorId, content },
      select: commentSelect,
    });

    await logAudit({
      orgId,
      userId: authorId,
      action: 'CREATED',
      entityType: 'comment',
      entityId: comment.id,
      after: { taskId, content },
      tx,
    });

    return comment;
  });
};

export const deleteComment = async (
  commentId: string,
  orgId: string,
  userId: string,
  userRole: string
) => {
  const comment = await prisma.comment.findUnique({
    where: { id: commentId },
    include: { task: true },
  });

  if (!comment) throw new NotFoundError('Comment');
  if (comment.task.orgId !== orgId) throw new ForbiddenError('Comment does not belong to this organization');

  // Authors can delete own comments; admins/owners can delete any
  const isAuthor = comment.authorId === userId;
  const isPrivileged = ['ADMIN', 'OWNER'].includes(userRole);
  if (!isAuthor && !isPrivileged) throw new ForbiddenError('Cannot delete another user\'s comment');

  await prisma.$transaction(async (tx) => {
    await tx.comment.delete({ where: { id: commentId } });

    await logAudit({
      orgId,
      userId,
      action: 'DELETED',
      entityType: 'comment',
      entityId: commentId,
      before: { taskId: comment.taskId, content: comment.content },
      tx,
    });
  });
};
```

---

4. Controller:

typescript

```typescript
// src/modules/comments/comment.controller.ts

import { Response, NextFunction } from 'express';
import { AuthRequest } from '../../types';
import { validateCreateCommentInput } from './comment.validation';
import { listComments, createComment, deleteComment } from './comment.service';

export const CommentController = {
  async list(req: AuthRequest, res: Response, next: NextFunction) {
    try {
      const page = parseInt(req.query.page as string || '1', 10);
      const limit = Math.min(parseInt(req.query.limit as string || '20', 10), 100);
      const result = await listComments(req.params.taskId, req.membership!.orgId, page, limit);
      res.json(result);
    } catch (err) { next(err); }
  },

  async create(req: AuthRequest, res: Response, next: NextFunction) {
    try {
      const { content } = validateCreateCommentInput(req.body);
      const comment = await createComment(
        req.params.taskId,
        req.membership!.orgId,
        req.user!.userId,
        content
      );
      res.status(201).json({ comment });
    } catch (err) { next(err); }
  },

  async delete(req: AuthRequest, res: Response, next: NextFunction) {
    try {
      await deleteComment(
        req.params.commentId,
        req.membership!.orgId,
        req.user!.userId,
        req.membership!.role
      );
      res.status(204).send();
    } catch (err) { next(err); }
  },
};
```

---

5. Routes:

typescript

```
// src/modules/comments/comment.routes.ts

import { Router } from 'express';
import { authenticate } from '../../middleware/auth';
import { resolveTenant } from '../../middleware/tenant';
import { requirePermission } from '../../middleware/permissions';
import { CommentController } from './comment.controller';
```

```
const router = Router({ mergeParams: true });

router.use(authenticate, resolveTenant);

router.get('/',             requirePermission('task:read'), CommentController.list);
router.post('/',            requirePermission('task:create'), CommentController.create);
router.delete('/:commentId',requirePermission('task:read'), CommentController.delete);

export { router as commentRoutes };
```

typescript

```
// src/app.ts — add route

app.use('/api/orgs/:orgId/tasks/:taskId/comments', commentRoutes);
```

Route map:

Method	**Route**	**Description**
GET	`/orgs/:orgId/tasks/:taskId/comments`	List comments
POST	`/orgs/:orgId/tasks/:taskId/comments`	Create comment
DELETE	`/orgs/:orgId/tasks/:taskId/comments/:commentId`	Delete comment

List response:

json

```
{
  "comments": [
    {
      "id": "uuid",
      "content": "Looks good, moving to review.",
      "createdAt": "2024-01-15T10:30:00.000Z",
      "updatedAt": "2024-01-15T10:30:00.000Z",
      "author": { "id": "uuid", "email": "jane@example.com" }
    }
  ],
  "pagination": { "total": 1, "page": 1, "limit": 20, "pages": 1 }
}
```

Key points:

`resolveTask` in service — both `listComments` and `createComment` verify the task exists, isn't soft-deleted, and belongs to the org before touching comments. No orphaned comments.

Ownership-aware delete — authors can delete their own comments; ADMINs and OWNERs can delete any. VIEWERs and MEMBERs can't delete others' comments. The check lives in the service, not middleware, because it requires knowing the comment's author.

`onDelete: Cascade` — deleting a task hard-deletes its comments. If you soft-delete tasks you may want to reconsider this — comments on a soft-deleted task will still exist until the task is hard-purged.

Audit logs comments — same `logAudit` pattern, `entityType: 'comment'`, so comment activity shows up in the org audit trail.

Ready to write tests or what's next?

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.

Feb 18

1. Schema:

prisma

```
// prisma/schema.prisma — add model

model Notification {
  id        String           @id @default(uuid())
  userId    String           @map("user_id")
  type      NotificationType
  title     String
  body      String
  meta      Json?
  readAt    DateTime?        @map("read_at")
  createdAt DateTime         @default(now()) @map("created_at")

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

  @@index([userId])
  @@index([userId, readAt])
  @@map("notifications")
}

enum NotificationType {
  TASK_ASSIGNED
  TASK_UPDATED
  COMMENT_ADDED
}
```

Add to `User` model: `notifications Notification[]`

bash

```
npx prisma migrate dev --name add_notifications
```

---

2. Notification service:

typescript

```
// src/modules/notifications/notification.service.ts

import { NotificationType } from '@prisma/client';
import { prisma } from '../../config/database';
import { NotFoundError, ForbiddenError } from '../../errors';

export const createNotification = async (
  userId: string,
  type: NotificationType,
  title: string,
  body: string,
  meta?: Record<string, unknown>,
  tx?: Parameters<typeof prisma.$transaction>[0] extends (tx: infer T) => unknown ? T : never
) => {
  const client = (tx ?? prisma) as typeof prisma;
  return client.notification.create({
    data: { userId, type, title, body, meta },
  });
};

export const notifyTaskAssigned = async (
  taskId: string,
  assignedToId: string,
  assignedByEmail: string,
  taskTitle: string,
  orgId: string,
  tx?: Parameters<typeof createNotification>[6]
) => {
  return createNotification(
    assignedToId,
    'TASK_ASSIGNED',
    'You have been assigned a task',
    `${assignedByEmail} assigned you "${taskTitle}"`,
    { taskId, orgId },
    tx
  );
};

export const listNotifications = async (
  userId: string,
  unreadOnly: boolean,
  page: number,
  limit: number
) => {
  const where = {
    userId,
    ...(unreadOnly && { readAt: null }),
  };

  const [notifications, total, unreadCount] = await
prisma.$transaction([
    prisma.notification.findMany({
      where,
      orderBy: { createdAt: 'desc' },
      skip: (page - 1) * limit,
      take: limit,
      select: {
        id: true,
        type: true,
        title: true,
        body: true,
        meta: true,
        readAt: true,
        createdAt: true,
      },
    }),
    prisma.notification.count({ where }),
    prisma.notification.count({ where: { userId, readAt: null } }),
  ]);

  return {
    notifications,
    unreadCount,
    pagination: { total, page, limit, pages: Math.ceil(total / limit)
},
  };
};

export const markAsRead = async (notificationId: string, userId:
string) => {
  const notification = await prisma.notification.findUnique({
    where: { id: notificationId },
  });

  if (!notification) throw new NotFoundError('Notification');
  if (notification.userId !== userId) throw new
ForbiddenError('Cannot mark another user\'s notification as read');
  if (notification.readAt) return notification; // already read,
no-op

  return prisma.notification.update({
    where: { id: notificationId },
    data: { readAt: new Date() },
    select: { id: true, readAt: true },
  });
};

export const markAllAsRead = async (userId: string) => {
  const { count } = await prisma.notification.updateMany({
    where: { userId, readAt: null },
    data: { readAt: new Date() },
  });

  return { marked: count };
};
```

---

3. Wire into task assignment:

typescript

```
// src/modules/tasks/task.service.ts — update createTask and
updateTask transactions

export const createTask = async (orgId: string, createdById: string,
data: CreateTaskInput) => {
  if (data.assignedToId) {
    const membership = await prisma.organizationMember.findUnique({
      where: { userId_orgId: { userId: data.assignedToId, orgId } },
    });
    if (!membership) throw new ForbiddenError('Assignee is not a member of this organization');
  }

  return prisma.$transaction(async (tx) => {
    const task = await tx.task.create({
      data: { ...data, orgId, createdById },
      select: taskSelect,
    });

    await logAudit({ orgId, userId: createdById, action: 'CREATED', entityType: 'task', entityId: task.id, after: task as Record<string, unknown>, tx });

    // Notify assignee if not self-assigned
    if (data.assignedToId && data.assignedToId !== createdById) {
      await notifyTaskAssigned(task.id, data.assignedToId, task.createdBy.email, task.title, orgId, tx);
    }

    return task;
  });
};

export const updateTask = async (taskId: string, orgId: string, userId: string, data: UpdateTaskInput) => {
  const existing = await getOwnedTask(taskId, orgId);

  if (data.assignedToId) {
    const membership = await prisma.organizationMember.findUnique({
      where: { userId_orgId: { userId: data.assignedToId, orgId } },
    });
    if (!membership) throw new ForbiddenError('Assignee is not a member of this organization');
  }

  return prisma.$transaction(async (tx) => {
    const task = await tx.task.update({
      where: { id: taskId },
      data,
      select: { ...taskSelect, createdBy: { select: { id: true, email: true } } },
    });

    const { before, after } = diffObjects(
      existing as unknown as Record<string, unknown>,
      task as unknown as Record<string, unknown>
    );

    await logAudit({ orgId, userId, action: 'UPDATED', entityType: 'task', entityId: taskId, before, after, tx });

    // Notify new assignee if assignment changed and not self-assigned
    const assigneeChanged = data.assignedToId && data.assignedToId !== existing.assignedToId;
    if (assigneeChanged && data.assignedToId !== userId) {
      await notifyTaskAssigned(task.id, data.assignedToId!, task.createdBy.email, task.title, orgId, tx);
    }

    return task;
  });
};
```

---

4. Controller:

typescript

```
// src/modules/notifications/notification.controller.ts

import { Response, NextFunction } from 'express';
import { AuthRequest } from '../../types';
import { listNotifications, markAsRead, markAllAsRead } from './notification.service';

export const NotificationController = {
  async list(req: AuthRequest, res: Response, next: NextFunction) {
    try {
      const unreadOnly = req.query.unreadOnly === 'true';
      const page  = parseInt(req.query.page  as string || '1',  10);
      const limit = Math.min(parseInt(req.query.limit as string || '20', 10), 100);

      const result = await listNotifications(req.user!.userId, unreadOnly, page, limit);
      res.json(result);
    } catch (err) { next(err); }
  },
  async markRead(req: AuthRequest, res: Response, next: NextFunction)
{
    try {
      const result = await markAsRead(req.params.notificationId,
req.user!.userId);
      res.json(result);
    } catch (err) { next(err); }
  },

  async markAllRead(req: AuthRequest, res: Response, next:
NextFunction) {
    try {
      const result = await markAllAsRead(req.user!.userId);
      res.json(result);
    } catch (err) { next(err); }
  },
};
```

---

5. Routes:

typescript

```
// src/modules/notifications/notification.routes.ts

import { Router } from 'express';
import { authenticate } from '../../middleware/auth';
import { NotificationController } from './notification.controller';

const router = Router();

router.use(authenticate);

router.get('/',
NotificationController.list);
router.patch('/:notificationId/read',
NotificationController.markRead);
router.patch('/read-all',
NotificationController.markAllRead);

export { router as notificationRoutes };
```

typescript

```
// src/app.ts — add route
app.use('/api/notifications', notificationRoutes);
```

Route map:

Method	Route	Description
GET	`/api/notifications`	List all notifications
GET	`/api/notifications?unreadOnly=true`	Unread only
PATCH	`/api/notifications/:id/read`	Mark one as read
PATCH	`/api/notifications/read-all`	Mark all as read

List response:

json

```json
{
  "notifications": [
    {
      "id": "uuid",
      "type": "TASK_ASSIGNED",
      "title": "You have been assigned a task",
      "body": "jane@example.com assigned you \"Build auth module\"",
      "meta": { "taskId": "uuid", "orgId": "uuid" },
      "readAt": null,
      "createdAt": "2024-01-15T10:30:00.000Z"
    }
  ],
  "unreadCount": 1,
  "pagination": { "total": 1, "page": 1, "limit": 20, "pages": 1 }
}
```

Key points:

Same transaction as task write — notification is created atomically with the task mutation. If either fails, both roll back. No phantom notifications for assignments that didn't persist.

No self-notification — skips creating a notification when the actor assigns the task to themselves.

Assignment change detection — `updateTask` only fires a notification when `assignedToId` actually changes, not on every update.

`unreadCount` always returned — even when not filtering by unread, the count comes back so clients can badge the notification icon without a second request.

`meta` JSON field — stores `taskId` and `orgId` so the client can build a deep link directly to the task without extra lookups.

Notifications are user-scoped not org-scoped — `/api/notifications` sits outside `/api/orgs/:orgId` because a user may belong to multiple orgs and wants a unified inbox.

Ready to write tests or what's next?

Add rate limiting to prevent API abuse. Registration and login endpoints should be more strictly limited than other endpoints.

Feb 18

bash

```
npm install express-rate-limit
```

1. Rate limit configs:

typescript

```
// src/config/rateLimit.ts

import rateLimit, { RateLimitRequestHandler } from
'express-rate-limit';

const rateLimitResponse = (retryAfter: number) => ({
  error: {
    code: 'RATE_LIMIT_EXCEEDED',
    message: 'Too many requests, please try again later',
    retryAfter,
    timestamp: new Date().toISOString(),
  },
});

const makeHandler = (
  windowMs: number,
  max: number,
  message?: string
): RateLimitRequestHandler =>
  rateLimit({
    windowMs,
    max,
    standardHeaders: true,    // RateLimit-* headers
    legacyHeaders: false,      // disable X-RateLimit-* headers
    handler: (req, res) => {
      const retryAfter = Math.ceil(windowMs / 1000);
      res.status(429).json(rateLimitResponse(retryAfter));
    },
    skip: () => process.env.NODE_ENV === 'test',
  });

// Auth endpoints — tightest limits
export const registrationLimiter = makeHandler(
  60 * 60 * 1000,  // 1 hour window
  10,              // 10 registrations per IP per hour
);

export const loginLimiter = makeHandler(
  15 * 60 * 1000,  // 15 min window
  20,              // 20 login attempts per IP per 15 min
);

// General API — loose limit, just prevents hammering
export const apiLimiter = makeHandler(
  60 * 1000,       // 1 min window
  100,             // 100 requests per IP per minute
);

// Notification/read endpoints — medium
export const readLimiter = makeHandler(
  60 * 1000,
  200,
);
```

---

2. Apply to routes:

typescript

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

import { registrationLimiter, loginLimiter } from
'../../config/rateLimit';

router.post('/register', registrationLimiter,
AuthController.register);
router.post('/login',    loginLimiter,        AuthController.login);
```

typescript

```
// src/app.ts

import { apiLimiter, readLimiter } from './config/rateLimit';

app.use('/api/notifications', readLimiter);
app.use('/api',               apiLimiter);   // catch-all after
specific limiters
```

```
app.use('/api/auth',   authRoutes);
app.use('/api/orgs',   orgRoutes);
app.use('/api/orgs/:orgId/tasks',                    taskRoutes);
app.use('/api/orgs/:orgId/tasks/:taskId/comments',   commentRoutes);
app.use('/api/orgs/:orgId/audit',                    auditRoutes);
app.use('/api/notifications',
notificationRoutes);

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

---

3. Add to error handler — rate limit errors are already handled by the `handler` callback in each limiter, but guard against anything slipping through:

typescript

~~~
// src/middleware/errorHandler.ts — add before unknown errors block

if (res.statusCode === 429) {
  return res.status(429).json(
    buildErrorResponse('RATE_LIMIT_EXCEEDED', 'Too many requests',
undefined, requestId)
  );
}
```

---

**Rate limit table:**

| Endpoint | Window | Max | Reasoning |
|----------|--------|-----|-----------|
| `POST /auth/register` | 1 hour | 10 | Prevents mass account
creation |
| `POST /auth/login` | 15 min | 20 | Limits brute force attempts |
| `GET /api/notifications` | 1 min | 200 | Read-heavy, looser limit |
| All other `/api/*` | 1 min | 100 | General abuse prevention |

**Rate limit headers on every response:**
```
RateLimit-Limit: 100
RateLimit-Remaining: 95
RateLimit-Reset: 1705312200
~~~

429 response:

json

```json
{
  "error": {
    "code": "RATE_LIMIT_EXCEEDED",
    "message": "Too many requests, please try again later",
    "retryAfter": 900,
    "timestamp": "2024-01-15T10:30:00.000Z"
  }
}
```

Key points:

Specific limiters before catch-all — `registrationLimiter` and `loginLimiter` are applied directly on their routes so they don't interfere with each other or the general `apiLimiter`.

`skip` in test env — rate limiting in tests causes flaky failures. The `skip` callback disables it when `NODE_ENV=test`.

`standardHeaders: true` — sends `RateLimit-*` headers per the IETF draft standard so clients know when to back off without parsing error bodies.

`retryAfter` in response — clients get the window in seconds so they can implement backoff without guessing.

Ready to write tests or what's next?

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.

Feb 18

bash

```bash
npm install -D jest ts-jest @types/jest
npx ts-jest config:init
```

typescript

```typescript
// jest.config.ts

export default {
  preset: 'ts-jest',
  testEnvironment: 'node',
  roots: ['<rootDir>/src'],
  testMatch: ['**/__tests__/**/*.test.ts'],
  clearMocks: true,
};
```

---

typescript

```typescript
// src/__tests__/middleware/auth.test.ts
```

```
import { Request, Response, NextFunction } from 'express';
import jwt from 'jsonwebtoken';
import { authenticate } from '../../middleware/auth';
import { AuthRequest, JWTPayload } from '../../types';

// Set secret before module loads
process.env.JWT_SECRET = 'test-secret';
process.env.DATABASE_URL = 'postgresql://test';

const mockPayload: JWTPayload = { userId: 'user-123', email: 'jane@example.com' };

const makeReq = (authHeader?: string): AuthRequest =>
  ({ headers: { authorization: authHeader } }) as AuthRequest;

const makeRes = (): Response => {
  const res = {} as Response;
  res.status = jest.fn().mockReturnValue(res);
  res.json   = jest.fn().mockReturnValue(res);
  return res;
};

const next: NextFunction = jest.fn();

const signToken = (payload: object, secret = 'test-secret', options?: jwt.SignOptions) =>
  jwt.sign(payload, secret, options);

describe('authenticate middleware', () => {
  beforeEach(() => jest.clearAllMocks());

  // ─── Missing / malformed header ────────────────────────────────────────────

  it('returns 401 when Authorization header is missing', () => {
    const req = makeReq();
    const res = makeRes();

    authenticate(req, res, next);

    expect(res.status).toHaveBeenCalledWith(401);
    expect(res.json).toHaveBeenCalledWith(
      expect.objectContaining({ error: 'Missing or invalid authorization header' })
    );
    expect(next).not.toHaveBeenCalled();
  });

  it('returns 401 when Authorization header has no Bearer prefix', () => {
    const req = makeReq('Basic sometoken');
    const res = makeRes();

    authenticate(req, res, next);

    expect(res.status).toHaveBeenCalledWith(401);
    expect(res.json).toHaveBeenCalledWith(
      expect.objectContaining({ error: 'Missing or invalid authorization header' })
    );
    expect(next).not.toHaveBeenCalled();
  });

  // ─── Invalid tokens ──────────────────────────────────────────────────────────

  it('returns 401 when token is signed with wrong secret', () => {
    const token = signToken(mockPayload, 'wrong-secret');
    const req   = makeReq(`Bearer ${token}`);
    const res   = makeRes();

    authenticate(req, res, next);

    expect(res.status).toHaveBeenCalledWith(401);
    expect(res.json).toHaveBeenCalledWith(
      expect.objectContaining({ error: 'Invalid token' })
    );
    expect(next).not.toHaveBeenCalled();
  });

  it('returns 401 when token is malformed', () => {
    const req = makeReq('Bearer not.a.valid.jwt');
    const res = makeRes();

    authenticate(req, res, next);

    expect(res.status).toHaveBeenCalledWith(401);
    expect(res.json).toHaveBeenCalledWith(
      expect.objectContaining({ error: 'Invalid token' })
    );
    expect(next).not.toHaveBeenCalled();
  });

  it('returns 401 with "Token expired" when token is expired', () => {
    const token = signToken(mockPayload, 'test-secret', { expiresIn: -1 });
    const req   = makeReq(`Bearer ${token}`);
    const res   = makeRes();

    authenticate(req, res, next);

    expect(res.status).toHaveBeenCalledWith(401);
    expect(res.json).toHaveBeenCalledWith(
      expect.objectContaining({ error: 'Token expired' })
    );
    expect(next).not.toHaveBeenCalled();
  });

  // ── Valid token ──────────────────────────────────────────────

  it('calls next() and attaches user to req when token is valid', () => {
    const token = signToken(mockPayload);
    const req   = makeReq(`Bearer ${token}`);
    const res   = makeRes();

    authenticate(req, res, next);

    expect(next).toHaveBeenCalledTimes(1);
    expect(next).toHaveBeenCalledWith(); // no error passed
    expect(req.user).toMatchObject(mockPayload);
    expect(res.status).not.toHaveBeenCalled();
  });

  it('attaches correct userId and email from token payload', () => {
    const payload: JWTPayload = { userId: 'abc-999', email: 'bob@example.com' };
    const token = signToken(payload);
    const req   = makeReq(`Bearer ${token}`);
    const res   = makeRes();

    authenticate(req, res, next);
    expect(req.user?.userId).toBe('abc-999');
    expect(req.user?.email).toBe('bob@example.com');
  });
});
```

---

typescript

```
// src/__tests__/middleware/tenant.test.ts

import { Response, NextFunction } from 'express';
import { resolveTenant } from '../../middleware/tenant';
import { AuthRequest } from '../../types';
import { prisma } from '../../config/database';

jest.mock('../../config/database', () => ({
  prisma: {
    organizationMember: { findUnique: jest.fn() },
  },
}));

const mockFindUnique = prisma.organizationMember.findUnique as
jest.Mock;

const makeReq = (orgId?: string, userId = 'user-123'): AuthRequest =>
  ({
    params: { orgId },
    user:   { userId, email: 'jane@example.com' },
  }) as unknown as AuthRequest;

const makeRes = (): Response => {
  const res = {} as Response;
  res.status = jest.fn().mockReturnValue(res);
  res.json   = jest.fn().mockReturnValue(res);
  return res;
};

const next: NextFunction = jest.fn();

describe('resolveTenant middleware', () => {
  beforeEach(() => jest.clearAllMocks());

  it('returns 400 when orgId param is missing', async () => {
    const req = makeReq(undefined);
    const res = makeRes();
    await resolveTenant(req, res, next);

    expect(res.status).toHaveBeenCalledWith(400);
    expect(res.json).toHaveBeenCalledWith(
      expect.objectContaining({ error: 'Missing orgId' })
    );
    expect(next).not.toHaveBeenCalled();
  });

  it('returns 403 when user is not a member of the org', async () => {
    mockFindUnique.mockResolvedValue(null);
    const req = makeReq('org-123');
    const res = makeRes();

    await resolveTenant(req, res, next);

    expect(res.status).toHaveBeenCalledWith(403);
    expect(res.json).toHaveBeenCalledWith(
      expect.objectContaining({ error: 'Access denied to this organization' })
    );
    expect(next).not.toHaveBeenCalled();
  });

  it('attaches membership and calls next() when user is a member', async () => {
    mockFindUnique.mockResolvedValue({ userId: 'user-123', orgId: 'org-123', role: 'MEMBER' });
    const req = makeReq('org-123');
    const res = makeRes();

    await resolveTenant(req, res, next);

    expect(next).toHaveBeenCalledWith();
    expect(req.membership).toEqual({ orgId: 'org-123', role: 'MEMBER' });
  });

  it('passes correct composite key to prisma lookup', async () => {
    mockFindUnique.mockResolvedValue({ userId: 'user-123', orgId: 'org-456', role: 'ADMIN' });
    const req = makeReq('org-456', 'user-123');
    const res = makeRes();
    await resolveTenant(req, res, next);

    expect(mockFindUnique).toHaveBeenCalledWith({
      where: { userId_orgId: { userId: 'user-123', orgId: 'org-456' }
},
    });
  });

  it('returns 500 on unexpected database error', async () => {
    mockFindUnique.mockRejectedValue(new Error('DB connection
lost'));
    const req = makeReq('org-123');
    const res = makeRes();

    await resolveTenant(req, res, next);

    expect(res.status).toHaveBeenCalledWith(500);
    expect(res.json).toHaveBeenCalledWith(
      expect.objectContaining({ error: 'Internal server error' })
    );
  });
});
```

---

typescript

```
// src/__tests__/middleware/permissions.test.ts

import { Response, NextFunction } from 'express';
import { requirePermission, can } from
'../../middleware/permissions';
import { AuthRequest } from '../../types';
import { Role } from '@prisma/client';

const makeReq = (role?: Role): AuthRequest =>
  ({ membership: role ? { orgId: 'org-123', role } : undefined }) as
AuthRequest;

const makeRes = (): Response => {
  const res = {} as Response;
  res.status = jest.fn().mockReturnValue(res);
  res.json   = jest.fn().mockReturnValue(res);
  return res;
};
```

```
const next: NextFunction = jest.fn();

describe('can() helper', () => {
  it.each([
    ['VIEWER', 'task:read',   true],
    ['VIEWER', 'task:create', false],
    ['MEMBER', 'task:create', true],
    ['MEMBER', 'task:delete', false],
    ['ADMIN',  'task:delete', true],
    ['ADMIN',  'org:delete',  false],
    ['OWNER',  'org:delete',  true],
  ] as [Role, Parameters<typeof can>[1], boolean][])(
    '%s can %s → %s',
    (role, action, expected) => {
      expect(can(role, action)).toBe(expected);
    }
  );
});

describe('requirePermission middleware', () => {
  beforeEach(() => jest.clearAllMocks());

  it('calls next() when user has sufficient role', () => {
    const req = makeReq('MEMBER');
    const res = makeRes();

    requirePermission('task:create')(req, res, next);

    expect(next).toHaveBeenCalledWith();
    expect(res.status).not.toHaveBeenCalled();
  });

  it('calls next(ForbiddenError) when user role is insufficient', ()
=> {
    const req = makeReq('VIEWER');
    const res = makeRes();

    requirePermission('task:create')(req, res, next);

    expect(next).toHaveBeenCalledWith(
      expect.objectContaining({ message: 'Missing permission:
task:create', statusCode: 403 })
    );
  });
  it('calls next(ForbiddenError) when membership is missing entirely', () => {
    const req = makeReq(undefined);
    const res = makeRes();

    requirePermission('task:read')(req, res, next);

    expect(next).toHaveBeenCalledWith(
      expect.objectContaining({ statusCode: 403 })
    );
  });

  it('OWNER passes all permission checks', () => {
    const actions: Parameters<typeof requirePermission>[0][] = [
      'task:read', 'task:create', 'task:update', 'task:delete',
      'member:add', 'member:remove', 'org:update', 'org:delete',
    ];

    actions.forEach((action) => {
      jest.clearAllMocks();
      const req = makeReq('OWNER');
      const res = makeRes();
      requirePermission(action)(req, res, next);
      expect(next).toHaveBeenCalledWith();
    });
  });
});
```

Run tests:

bash

~~~
npx jest --coverage
```

**Coverage output:**
```
PASS src/__tests__/middleware/auth.test.ts
PASS src/__tests__/middleware/tenant.test.ts
PASS src/__tests__/middleware/permissions.test.ts

Coverage summary:
  middleware/auth.ts        100% statements
  middleware/tenant.ts      100% statements
  middleware/permissions.ts 100% statements
~~~

Key points:

No real DB in unit tests — `prisma` is fully mocked with `jest.mock`. Tests run in milliseconds with no external dependencies.

`it.each` for permission matrix — the `can()` helper is tested across all role/action combos in one table, making it easy to add new cases.

Expired token test — `expiresIn: -1` issues an already-expired token, cleanly testing the `TokenExpiredError` branch without `setTimeout`.

`next` called with no args vs error — `toHaveBeenCalledWith()` (no args) verifies the happy path; `toHaveBeenCalledWith(expect.objectContaining({statusCode: 403}))` verifies errors are forwarded correctly rather than swallowed.

`clearMocks: true` in jest config — resets all mock state between tests automatically so call counts don't bleed across cases.

Ready to add integration tests or what's next?

Write integration tests for the task creation endpoint. Test:
- Successful task creation
- Unauthorized access (no token)
- Forbidden access (user not in organization)

Feb 18

bash

```
npm install -D supertest @types/supertest
```

typescript

```
// src/__tests__/helpers/testApp.ts

import express from 'express';
import { taskRoutes } from '../../modules/tasks/task.routes';
import { authRoutes } from '../../modules/auth/auth.routes';
import { orgRoutes } from '../../modules/orgs/org.routes';
import { errorHandler } from '../../middleware/errorHandler';
import { notFound } from '../../middleware/notFound';

export const buildTestApp = () => {
  const app = express();
  app.use(express.json());
  app.use('/api/auth', authRoutes);
  app.use('/api/orgs', orgRoutes);
  app.use('/api/orgs/:orgId/tasks', taskRoutes);
  app.use(notFound);
  app.use(errorHandler);
  return app;
};
```

typescript

```typescript
// src/__tests__/helpers/db.ts

import { prisma } from '../../config/database';

export const cleanDb = async () => {
  // Order matters — delete children before parents
  await prisma.auditLog.deleteMany();
  await prisma.notification.deleteMany();
  await prisma.comment.deleteMany();
  await prisma.task.deleteMany();
  await prisma.organizationMember.deleteMany();
  await prisma.organization.deleteMany();
  await prisma.user.deleteMany();
};
```

typescript

```typescript
// src/__tests__/helpers/factories.ts

import bcrypt from 'bcrypt';
import jwt from 'jsonwebtoken';
import { Role } from '@prisma/client';
import { prisma } from '../../config/database';
import { env } from '../../config/env';

export const createUser = async (email = 'test@example.com', password = 'Password1') => {
  const passwordHash = await bcrypt.hash(password, 10);
  return prisma.user.create({ data: { email, passwordHash } });
};

export const createOrg = async (slug = 'test-org') => {
  return prisma.organization.create({ data: { name: 'Test Org', slug } });
};

export const addToOrg = async (userId: string, orgId: string, role: Role = 'MEMBER') => {
  return prisma.organizationMember.create({ data: { userId, orgId, role } });
};

export const tokenFor = (userId: string, email: string) =>
```

```
  jwt.sign({ userId, email }, env.JWT_SECRET, { expiresIn: '1h' });
```

typescript

```typescript
// src/__tests__/integration/tasks/createTask.test.ts

import request from 'supertest';
import { buildTestApp } from '../../helpers/testApp';
import { cleanDb } from '../../helpers/db';
import { createUser, createOrg, addToOrg, tokenFor } from
'../../helpers/factories';
import { prisma } from '../../../config/database';

const app = buildTestApp();

beforeAll(() => cleanDb());
afterEach(() => cleanDb());
afterAll(() => prisma.$disconnect());

// ─── Fixtures
```

```typescript
const validBody = {
  title: 'Build login page',
  description: 'OAuth + email/password',
  status: 'TODO',
  priority: 'HIGH',
};

// ─── Success
```

```typescript
describe('POST /api/orgs/:orgId/tasks — success', () => {
  it('creates a task and returns 201 with task shape', async () => {
    const user  = await createUser();
    const org   = await createOrg();
    await addToOrg(user.id, org.id, 'MEMBER');
    const token = tokenFor(user.id, user.email);

    const res = await request(app)
      .post(`/api/orgs/${org.id}/tasks`)
      .set('Authorization', `Bearer ${token}`)
      .send(validBody);

    expect(res.status).toBe(201);
    expect(res.body.task).toMatchObject({
      title:       'Build login page',
      description: 'OAuth + email/password',
      status:      'TODO',
      priority:    'HIGH',
    });
    expect(res.body.task.id).toBeDefined();
    expect(res.body.task.createdBy.email).toBe(user.email);
  });

  it('persists the task in the database', async () => {
    const user  = await createUser();
    const org   = await createOrg();
    await addToOrg(user.id, org.id);
    const token = tokenFor(user.id, user.email);

    const res = await request(app)
      .post(`/api/orgs/${org.id}/tasks`)
      .set('Authorization', `Bearer ${token}`)
      .send(validBody);

    const saved = await prisma.task.findUnique({ where: { id:
res.body.task.id } });
    expect(saved).not.toBeNull();
    expect(saved?.title).toBe('Build login page');
    expect(saved?.orgId).toBe(org.id);
    expect(saved?.createdById).toBe(user.id);
  });

  it('creates an audit log entry on task creation', async () => {
    const user  = await createUser();
    const org   = await createOrg();
    await addToOrg(user.id, org.id);
    const token = tokenFor(user.id, user.email);

    const res = await request(app)
      .post(`/api/orgs/${org.id}/tasks`)
      .set('Authorization', `Bearer ${token}`)
      .send(validBody);

    const log = await prisma.auditLog.findFirst({
      where: { entityId: res.body.task.id, action: 'CREATED' },
    });
    expect(log).not.toBeNull();
    expect(log?.userId).toBe(user.id);
    expect(log?.orgId).toBe(org.id);
  });

  it('creates a notification when task is assigned to another user', async () => {
    const creator  = await createUser('creator@example.com');
    const assignee = await createUser('assignee@example.com');
    const org      = await createOrg();
    await addToOrg(creator.id,  org.id, 'MEMBER');
    await addToOrg(assignee.id, org.id, 'MEMBER');
    const token = tokenFor(creator.id, creator.email);

    await request(app)
      .post(`/api/orgs/${org.id}/tasks`)
      .set('Authorization', `Bearer ${token}`)
      .send({ ...validBody, assignedToId: assignee.id });

    const notification = await prisma.notification.findFirst({
      where: { userId: assignee.id, type: 'TASK_ASSIGNED' },
    });
    expect(notification).not.toBeNull();
    expect(notification?.body).toContain(creator.email);
  });

  it('does not create a notification when self-assigning', async () => {
    const user  = await createUser();
    const org   = await createOrg();
    await addToOrg(user.id, org.id);
    const token = tokenFor(user.id, user.email);

    await request(app)
      .post(`/api/orgs/${org.id}/tasks`)
      .set('Authorization', `Bearer ${token}`)
      .send({ ...validBody, assignedToId: user.id });

    const notifications = await prisma.notification.findMany({
      where: { userId: user.id },
    });
    expect(notifications).toHaveLength(0);
  });

  it('defaults status to TODO and priority to MEDIUM when omitted', async () => {
    const user  = await createUser();
    const org   = await createOrg();
    await addToOrg(user.id, org.id);
    const token = tokenFor(user.id, user.email);

    const res = await request(app)
      .post(`/api/orgs/${org.id}/tasks`)
      .set('Authorization', `Bearer ${token}`)
      .send({ title: 'Minimal task' });

    expect(res.status).toBe(201);
    expect(res.body.task.status).toBe('TODO');
    expect(res.body.task.priority).toBe('MEDIUM');
  });
});

// ─── Unauthorized
─────────────────────────────────────────────────────────────

describe('POST /api/orgs/:orgId/tasks — unauthorized', () => {
  it('returns 401 when no token is provided', async () => {
    const org = await createOrg();

    const res = await request(app)
      .post(`/api/orgs/${org.id}/tasks`)
      .send(validBody);

    expect(res.status).toBe(401);
    expect(res.body.error).toMatchObject({ code: 'UNAUTHORIZED' });
  });

  it('returns 401 when token is malformed', async () => {
    const org = await createOrg();

    const res = await request(app)
      .post(`/api/orgs/${org.id}/tasks`)
      .set('Authorization', 'Bearer not.a.real.token')
      .send(validBody);

    expect(res.status).toBe(401);
    expect(res.body.error).toMatchObject({ code: 'UNAUTHORIZED' });
  });

  it('returns 401 when token is expired', async () => {
    const user  = await createUser();
    const org   = await createOrg();
    const token = jwt.sign(
      { userId: user.id, email: user.email },
      env.JWT_SECRET,
      { expiresIn: -1 }
    );

    const res = await request(app)
      .post(`/api/orgs/${org.id}/tasks`)
      .set('Authorization', `Bearer ${token}`)
      .send(validBody);

    expect(res.status).toBe(401);
  });
});

// —— Forbidden
```

```
describe('POST /api/orgs/:orgId/tasks — forbidden', () => {
  it('returns 403 when user is not a member of the org', async () => {
    const user  = await createUser();
    const org   = await createOrg();
    // deliberately not adding user to org
    const token = tokenFor(user.id, user.email);

    const res = await request(app)
      .post(`/api/orgs/${org.id}/tasks`)
      .set('Authorization', `Bearer ${token}`)
      .send(validBody);

    expect(res.status).toBe(403);
    expect(res.body.error).toMatchObject({
      code: 'FORBIDDEN',
      message: 'Access denied to this organization',
    });
  });

  it('returns 403 when VIEWER tries to create a task', async () => {
    const user  = await createUser();
    const org   = await createOrg();
    await addToOrg(user.id, org.id, 'VIEWER');
    const token = tokenFor(user.id, user.email);

    const res = await request(app)
      .post(`/api/orgs/${org.id}/tasks`)
      .set('Authorization', `Bearer ${token}`)
      .send(validBody);

    expect(res.status).toBe(403);
    expect(res.body.error).toMatchObject({ code: 'FORBIDDEN' });
  });

  it('returns 403 when assigning to a user outside the org', async ()
=> {
    const user    = await createUser('member@example.com');
    const outsider = await createUser('outsider@example.com');
    const org     = await createOrg();
    await addToOrg(user.id, org.id, 'MEMBER');
    const token = tokenFor(user.id, user.email);

    const res = await request(app)
      .post(`/api/orgs/${org.id}/tasks`)
      .set('Authorization', `Bearer ${token}`)
      .send({ ...validBody, assignedToId: outsider.id });

    expect(res.status).toBe(403);
    expect(res.body.error.message).toContain('not a member');
  });

  it('returns 404 when org does not exist', async () => {
    const user  = await createUser();
    const token = tokenFor(user.id, user.email);

    const res = await request(app)
      .post('/api/orgs/non-existent-org-id/tasks')
      .set('Authorization', `Bearer ${token}`)
      .send(validBody);

    expect(res.status).toBe(403); // resolveTenant returns 403, not
404 (don't leak org existence)
  });
});

// ── Validation
──────────────────────────────────────────────────────────────

describe('POST /api/orgs/:orgId/tasks — validation', () => {
  it('returns 422 when title is missing', async () => {
    const user  = await createUser();
    const org   = await createOrg();
    await addToOrg(user.id, org.id);
    const token = tokenFor(user.id, user.email);

    const res = await request(app)
      .post(`/api/orgs/${org.id}/tasks`)
      .set('Authorization', `Bearer ${token}`)
      .send({ status: 'TODO' });

    expect(res.status).toBe(422);
    expect(res.body.error.code).toBe('VALIDATION_ERROR');
    expect(res.body.error.details).toEqual(
      expect.arrayContaining([expect.stringContaining('title')])
    );
  });

  it('returns 422 when status is invalid enum value', async () => {
    const user   = await createUser();
    const org    = await createOrg();
    await addToOrg(user.id, org.id);
    const token = tokenFor(user.id, user.email);

    const res = await request(app)
      .post(`/api/orgs/${org.id}/tasks`)
      .set('Authorization', `Bearer ${token}`)
      .send({ ...validBody, status: 'INVALID_STATUS' });

    expect(res.status).toBe(422);
    expect(res.body.error.details).toEqual(
      expect.arrayContaining([expect.stringContaining('status')])
    );
  });
});
```

typescript

```typescript
// src/__tests__/setup.ts

import { cleanDb } from './helpers/db';
import { prisma } from '../config/database';

beforeAll(() => cleanDb());
afterAll(() => prisma.$disconnect());
```

typescript

```typescript
// jest.config.ts — update for integration tests
```

```
export default {
  preset: 'ts-jest',
  testEnvironment: 'node',
  roots: ['<rootDir>/src'],
  testMatch: ['**/__tests__/**/*.test.ts'],
  globalSetup: './src/__tests__/setup.ts',
  clearMocks: true,
  testTimeout: 15000,    // DB ops need more time
};
```

.env.test:

```env
DATABASE_URL="postgresql://postgres:postgres@localhost:5432/taskapi_test"
JWT_SECRET="test-secret-integration"
NODE_ENV="test"
```

Run integration tests only:

```bash
NODE_ENV=test npx jest --testPathPattern=integration --runInBand
```

`--runInBand` runs tests serially — essential when tests share a database to prevent race conditions between `cleanDb` calls.

Key points:

Real DB, no mocks — integration tests hit an actual Postgres instance with real Prisma queries. This catches issues unit tests miss: constraint violations, transaction rollbacks, cascade deletes.

`cleanDb` order — deletes children before parents to avoid FK constraint failures. `afterEach` keeps tests isolated; `beforeAll` handles a dirty DB from a previous crashed run.

Factories not fixtures — `createUser`, `createOrg`, `addToOrg` build only what each test needs. No shared global state between tests.

403 for nonexistent org — `resolveTenant` returns 403 not 404 for orgs the user isn't a member of. This avoids leaking whether an org exists to users who shouldn't know.

Side effect assertions — success tests don't just check the response; they query the DB directly to verify the task was persisted, the audit log was written, and notifications were created.

`--runInBand` — integration tests must run serially when sharing a test database. Parallel runs cause `cleanDb` in one test to wipe data another test is reading.

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.

Feb 18

bash

```bash
npm install swagger-ui-express
npm install -D @types/swagger-ui-express
```

1. OpenAPI spec — base:

typescript

```typescript
// src/docs/openapi.ts

export const openApiSpec = {
  openapi: '3.0.3',
  info: {
    title: 'Task Management API',
    version: '1.0.0',
    description: 'Multi-tenant task management API with
organization-based access control',
  },
  servers: [
    { url: '/api', description: 'API base' },
  ],
  components: {
    securitySchemes: {
      bearerAuth: {
        type: 'http',
        scheme: 'bearer',
        bearerFormat: 'JWT',
      },
    },
    schemas: {
      Error: {
        type: 'object',
        properties: {
          error: {
            type: 'object',
            properties: {
              code:      { type: 'string', example: 'NOT_FOUND' },
              message:   { type: 'string', example: 'Task not found'
},
              details:   { type: 'array', items: { type: 'string' },
nullable: true },
              requestId: { type: 'string', nullable: true },
              timestamp: { type: 'string', format: 'date-time' },
            },
          },
        },
      },
      Pagination: {
        type: 'object',
        properties: {
          total: { type: 'integer', example: 42 },
          page:  { type: 'integer', example: 1  },
          limit: { type: 'integer', example: 20 },
          pages: { type: 'integer', example: 3  },
        },
      },
      User: {
        type: 'object',
        properties: {
          id:    { type: 'string', format: 'uuid' },
          email: { type: 'string', format: 'email' },
        },
      },
      TaskStatus: {
        type: 'string',
        enum: ['TODO', 'IN_PROGRESS', 'DONE'],
      },
      Priority: {
        type: 'string',
        enum: ['LOW', 'MEDIUM', 'HIGH'],
      },
      Task: {
        type: 'object',
        properties: {
          id:          { type: 'string', format: 'uuid' },
          title:       { type: 'string', example: 'Build login page'
},
          description: { type: 'string', nullable: true },
          status:      { $ref: '#/components/schemas/TaskStatus' },
          priority:    { $ref: '#/components/schemas/Priority' },
          dueDate:     { type: 'string', format: 'date-time',
nullable: true },
          createdAt:   { type: 'string', format: 'date-time' },
          updatedAt:   { type: 'string', format: 'date-time' },
          createdBy:   { $ref: '#/components/schemas/User' },
          assignedTo:  { allOf: [{ $ref: '#/components/schemas/User'
}], nullable: true },
        },
      },
      CreateTaskBody: {
        type: 'object',
        required: ['title'],
        properties: {
          title:       { type: 'string', minLength: 1, maxLength:
500, example: 'Build login page' },
          description: { type: 'string', example: 'OAuth +
email/password' },
          status:      { $ref: '#/components/schemas/TaskStatus' },
          priority:    { $ref: '#/components/schemas/Priority' },
          assignedToId:{ type: 'string', format: 'uuid' },
          dueDate:     { type: 'string', format: 'date-time' },
        },
      },
      UpdateTaskBody: {
        type: 'object',
        properties: {
          title:       { type: 'string', minLength: 1, maxLength: 500
},
          description: { type: 'string', nullable: true },
          status:      { $ref: '#/components/schemas/TaskStatus' },
          priority:    { $ref: '#/components/schemas/Priority' },
          assignedToId:{ type: 'string', format: 'uuid', nullable:
true },
          dueDate:     { type: 'string', format: 'date-time',
nullable: true },
        },
      },
    },
    parameters: {
      orgId: {
        name: 'orgId', in: 'path', required: true,
        schema: { type: 'string', format: 'uuid' },
        description: 'Organization ID',
      },
      taskId: {
        name: 'taskId', in: 'path', required: true,
        schema: { type: 'string', format: 'uuid' },
        description: 'Task ID',
      },
      page: {
        name: 'page', in: 'query', required: false,
        schema: { type: 'integer', minimum: 1, default: 1 },
      },
      limit: {
        name: 'limit', in: 'query', required: false,
        schema: { type: 'integer', minimum: 1, maximum: 100, default: 20 },
      },
    },
    responses: {
      Unauthorized: {
        description: 'Missing or invalid JWT token',
        content: { 'application/json': { schema: { $ref: '#/components/schemas/Error' } } },
      },
      Forbidden: {
        description: 'Insufficient permissions',
        content: { 'application/json': { schema: { $ref: '#/components/schemas/Error' } } },
      },
      NotFound: {
        description: 'Resource not found',
        content: { 'application/json': { schema: { $ref: '#/components/schemas/Error' } } },
      },
      ValidationError: {
        description: 'Invalid request body',
        content: { 'application/json': { schema: { $ref: '#/components/schemas/Error' } } },
      },
    },
  },
  security: [{ bearerAuth: [] }],
  paths: {},
};
```

---

2. Task path definitions:

typescript

```typescript
// src/docs/paths/tasks.ts

export const taskPaths = {
  '/orgs/{orgId}/tasks': {
    parameters: [{ $ref: '#/components/parameters/orgId' }],

    get: {
      summary: 'List tasks',
      description: 'Returns paginated tasks for an organization.
Requires VIEWER role or above.',
      operationId: 'listTasks',
      tags: ['Tasks'],
      parameters: [
        { $ref: '#/components/parameters/page' },
        { $ref: '#/components/parameters/limit' },
        {
          name: 'status', in: 'query',
          schema: { $ref: '#/components/schemas/TaskStatus' },
          description: 'Filter by status',
        },
        {
          name: 'priority', in: 'query',
          schema: { $ref: '#/components/schemas/Priority' },
          description: 'Filter by priority',
        },
        {
          name: 'assignedToId', in: 'query',
          schema: { type: 'string', format: 'uuid' },
          description: 'Filter by assignee',
        },
        {
          name: 'dueDateFrom', in: 'query',
          schema: { type: 'string', format: 'date-time' },
          description: 'Filter tasks due on or after this date',
        },
        {
          name: 'dueDateTo', in: 'query',
          schema: { type: 'string', format: 'date-time' },
          description: 'Filter tasks due on or before this date',
        },
      ],
      responses: {
        200: {
          description: 'Paginated task list',
          content: {
            'application/json': {
              schema: {
                type: 'object',
                properties: {
                  tasks:      { type: 'array', items: { $ref:
'#/components/schemas/Task' } },
                  pagination: { $ref:
'#/components/schemas/Pagination' },
                },
              },
              example: {
                tasks: [
                  {
                    id: 'uuid', title: 'Build login page', status:
'IN_PROGRESS',
                    priority: 'HIGH', dueDate:
'2024-02-01T00:00:00.000Z',
                    createdAt: '2024-01-15T10:30:00.000Z', updatedAt:
'2024-01-15T10:30:00.000Z',
                    createdBy: { id: 'uuid', email:
'jane@example.com' },
                    assignedTo: { id: 'uuid', email:
'john@example.com' },
                  },
                ],
                pagination: { total: 42, page: 1, limit: 20, pages: 3
},
              },
            },
          },
        },
        401: { $ref: '#/components/responses/Unauthorized' },
        403: { $ref: '#/components/responses/Forbidden' },
      },
    },

    post: {
      summary: 'Create a task',
      description: 'Creates a new task in the organization. Requires
MEMBER role or above.',
      operationId: 'createTask',
      tags: ['Tasks'],
      requestBody: {
        required: true,
        content: {
          'application/json': {
            schema: { $ref: '#/components/schemas/CreateTaskBody' },
            example: {
              title: 'Build login page',
              description: 'OAuth + email/password',
              status: 'TODO',
              priority: 'HIGH',
              dueDate: '2024-02-01T00:00:00.000Z',
            },
          },
        },
      },
      responses: {
        201: {
          description: 'Task created',
          content: {
            'application/json': {
              schema: {
                type: 'object',
                properties: { task: { $ref:
'#/components/schemas/Task' } },
              },
            },
          },
        },
        401: { $ref: '#/components/responses/Unauthorized' },
        403: { $ref: '#/components/responses/Forbidden' },
        422: { $ref: '#/components/responses/ValidationError' },
      },
    },
  },

  '/orgs/{orgId}/tasks/{taskId}': {
    parameters: [
      { $ref: '#/components/parameters/orgId' },
      { $ref: '#/components/parameters/taskId' },
    ],

    get: {
      summary: 'Get a task',
      operationId: 'getTask',
      tags: ['Tasks'],
      responses: {
        200: {
          description: 'Task detail',
          content: {
            'application/json': {
              schema: {
                type: 'object',
                properties: { task: { $ref:
'#/components/schemas/Task' } },
              },
            },
          },
        },
        401: { $ref: '#/components/responses/Unauthorized' },
        403: { $ref: '#/components/responses/Forbidden' },
        404: { $ref: '#/components/responses/NotFound' },
      },
    },

    patch: {
      summary: 'Update a task',
      description: 'Partially updates a task. Requires MEMBER role or
above.',
      operationId: 'updateTask',
      tags: ['Tasks'],
      requestBody: {
        required: true,
        content: {
          'application/json': {
            schema: { $ref: '#/components/schemas/UpdateTaskBody' },
          },
        },
      },
      responses: {
        200: {
          description: 'Updated task',
          content: {
            'application/json': {
              schema: {
                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: {
      summary: 'Delete a task',
      description: 'Soft-deletes a task. Requires ADMIN role or
above.',
      operationId: 'deleteTask',
      tags: ['Tasks'],
      responses: {
        204: { description: 'Task deleted' },
        401: { $ref: '#/components/responses/Unauthorized' },
        403: { $ref: '#/components/responses/Forbidden' },
        404: { $ref: '#/components/responses/NotFound' },
      },
    },
  },

  '/orgs/{orgId}/tasks/{taskId}/restore': {
    parameters: [
      { $ref: '#/components/parameters/orgId' },
      { $ref: '#/components/parameters/taskId' },
    ],
    post: {
      summary: 'Restore a deleted task',
      description: 'Restores a soft-deleted task. Requires ADMIN role
or above.',
      operationId: 'restoreTask',
      tags: ['Tasks'],
      responses: {
        200: {
          description: 'Restored task',
          content: {
            'application/json': {
              schema: {
                type: 'object',
                properties: { task: { $ref:
'#/components/schemas/Task' } },
              },
            },
          },
        },
        400: { description: 'Task is not deleted' },
        401: { $ref: '#/components/responses/Unauthorized' },
        403: { $ref: '#/components/responses/Forbidden' },
        404: { $ref: '#/components/responses/NotFound' },
      },
    },
  },
  '/orgs/{orgId}/tasks/deleted': {
    parameters: [{ $ref: '#/components/parameters/orgId' }],
    get: {
      summary: 'List deleted tasks',
      description: 'Returns soft-deleted tasks. Requires VIEWER role or above.',
      operationId: 'listDeletedTasks',
      tags: ['Tasks'],
      parameters: [
        { $ref: '#/components/parameters/page' },
        { $ref: '#/components/parameters/limit' },
      ],
      responses: {
        200: {
          description: 'Paginated deleted tasks',
          content: {
            'application/json': {
              schema: {
                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' },
      },
    },
  },
};
```

---

3. Assemble and serve:

typescript

```
// src/docs/index.ts

import { openApiSpec } from './openapi';
import { taskPaths } from './paths/tasks';

export const buildSpec = () => ({
```

~~~~
  ...openApiSpec,
  paths: {
    ...taskPaths,
  },
});
~~~~

typescript

~~~~
// src/app.ts — add docs route

import swaggerUi from 'swagger-ui-express';
import { buildSpec } from './docs';

const spec = buildSpec();

app.use('/api/docs', swaggerUi.serve, swaggerUi.setup(spec, {
  customSiteTitle: 'Task API Docs',
  swaggerOptions: { persistAuthorization: true },
}));

// Expose raw spec for tooling (Postman import, codegen, etc.)
app.get('/api/docs.json', (req, res) => res.json(spec));
```

---

**Docs available at:**
```
http://localhost:3000/api/docs      ← Swagger UI
http://localhost:3000/api/docs.json ← Raw OpenAPI JSON
```

**Folder structure:**
```
src/docs/
├── index.ts          ← assembles full spec
├── openapi.ts        ← base spec, components, schemas
└── paths/
    └── tasks.ts      ← task path definitions
    # auth.ts, orgs.ts, notifications.ts added as you go
~~~~

Key points:

`$ref` everywhere — schemas, parameters, and responses are defined once in `components` and referenced throughout. Adding a field to `Task` updates every endpoint automatically.

`persistAuthorization: true` — the Swagger UI remembers your JWT between page refreshes so you don't re-paste the token constantly during development.

`/api/docs.json` endpoint — exposes the raw spec for Postman collection import, client SDK generation (`openapi-generator`), or contract testing.

Path-per-file — `paths/tasks.ts`, `paths/auth.ts` etc. keep each module's docs next to its code conceptually. The `index.ts` assembles them without any module knowing about the others.

`operationId` — unique per endpoint, used by code generators to name SDK methods (`tasksApi.createTask(...)` etc.).

Ready to add anything else or wrap up?

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.

Feb 18

Good question. Here's an honest review of what we built across this conversation.

---

1. `userId` missing from `updateTask` / `deleteTask` controller signatures

We added `userId` as a parameter to the service in the audit logging step, but never updated the controller calls consistently.

typescript

```typescript
// Current — missing userId in some controllers
const task = await updateTask(req.params.taskId,
req.membership!.orgId, data);

// Should be everywhere
const task = await updateTask(req.params.taskId,
req.membership!.orgId, req.user!.userId, data);
```

---

2. Pagination validation is inconsistent

Task list uses Zod via `validateTaskListQuery`. Comment list and notification list parse `page`/`limit` inline with raw `parseInt` and no validation schema.

typescript

```typescript
// comment.controller.ts and notification.controller.ts — raw parsing
const page  = parseInt(req.query.page as string || '1', 10);
const limit = Math.min(parseInt(req.query.limit as string || '20',
10), 100);
```

Fix — extract a shared Zod schema:

typescript

```
// src/lib/pagination.ts

import { z } from 'zod';
import { validate } from './validate';

export const paginationSchema = z.object({
  page:  z.coerce.number().int().min(1).default(1),
  limit: z.coerce.number().int().min(1).max(100).default(20),
});

export type PaginationInput = z.infer<typeof paginationSchema>;

export const validatePagination = (query: unknown): PaginationInput
=>
  validate(paginationSchema, query);
```

Then in every list controller:

typescript

```
const { page, limit } = validatePagination(req.query);
```

---

3. `createNotification` has a broken `tx` type

typescript

```
// Current — overly complex inline type that won't work cleanly
tx?: Parameters<typeof prisma.$transaction>[0] extends (tx: infer T)
=> unknown ? T : never
```

Fix — use Prisma's exported type:

typescript

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

export const createNotification = async (
  userId: string,
  type: NotificationType,
  title: string,
  body: string,
  meta?: Record<string, unknown>,
  tx?: Prisma.TransactionClient      // clean, correct type
) => {
  const client = tx ?? prisma;
  return client.notification.create({ data: { userId, type, title,
body, meta } });
};
```

---

4. `logAudit` and `createNotification` have inconsistent `tx` patterns

`logAudit` accepts `tx` as part of a params object. `createNotification` accepts it as a positional argument. Pick one.

typescript

```
// logAudit — named params object ✅
await logAudit({ orgId, userId, action, entityType, entityId, before, after, tx });

// createNotification — positional ❌
await createNotification(userId, type, title, body, meta, tx);

// Fix createNotification to match
interface CreateNotificationParams {
  userId: string;
  type: NotificationType;
  title: string;
  body: string;
  meta?: Record<string, unknown>;
  tx?: Prisma.TransactionClient;
}

export const createNotification = async ({
  userId, type, title, body, meta, tx,
}: CreateNotificationParams) => { ... };
```

---

5. `deleteComment` checks role as a raw string

typescript

```
// comment.service.ts — bypasses the permissions system entirely
const isPrivileged = ['ADMIN', 'OWNER'].includes(userRole);
```

This duplicates role logic that already lives in `permissions.ts`. Fix — use the `can()` helper:

typescript

```
import { can } from '../../middleware/permissions';
import { Role } from '@prisma/client';

const isAuthor     = comment.authorId === userId;
const isPrivileged = can(userRole as Role, 'task:delete');

if (!isAuthor && !isPrivileged) {
  throw new ForbiddenError('Cannot delete another user\'s comment');
}
```

---

6. `addMember` service defines its own `roleHierarchy`

typescript

```typescript
// org.service.ts — duplicate of the hierarchy in permissions.ts
const roleHierarchy: Record<Role, number> = {
  OWNER: 4, ADMIN: 3, MEMBER: 2, VIEWER: 1,
};
```

Move the hierarchy to a shared location and import it:

typescript

```typescript
// src/lib/roles.ts

import { Role } from '@prisma/client';

export const roleHierarchy: Record<Role, number> = {
  OWNER: 4,
  ADMIN: 3,
  MEMBER: 2,
  VIEWER: 1,
};

export const roleExceeds = (actor: Role, target: Role): boolean =>
  roleHierarchy[actor] > roleHierarchy[target];
```

typescript

```typescript
// permissions.ts and org.service.ts both import from here
import { roleHierarchy, roleExceeds } from '../lib/roles';
```

---

7. `addMember` validation blocks VIEWER from being assigned

typescript

```typescript
// org.validation.ts
const addMemberSchema = z.object({
  role: z.enum(['ADMIN', 'MEMBER', 'VIEWER'] as
const).default('MEMBER'),
});
```

This is correct — but the original validation before Zod was:

typescript

```typescript
const validRoles: Role[] = ['ADMIN', 'MEMBER']; // VIEWER missing
```

If you kept the pre-Zod version anywhere, VIEWER can never be assigned. Confirm the Zod version is the only one in use.

8. `notifyTaskAssigned` has a stale 6-arg positional signature

After fixing `createNotification` to use a params object, `notifyTaskAssigned` needs updating:

typescript

```typescript
// src/modules/notifications/notification.service.ts

export const notifyTaskAssigned = async (
  taskId: string,
  assignedToId: string,
  assignedByEmail: string,
  taskTitle: string,
  orgId: string,
  tx?: Prisma.TransactionClient
) =>
  createNotification({
    userId: assignedToId,
    type: 'TASK_ASSIGNED',
    title: 'You have been assigned a task',
    body: `${assignedByEmail} assigned you "${taskTitle}"`,
    meta: { taskId, orgId },
    tx,
  });
```

9. Error handler checks `res.statusCode === 429` incorrectly

typescript

```typescript
// errorHandler.ts — this block is unreachable
// express-rate-limit calls res.status(429).json() directly, never calls next(err)
if (res.statusCode === 429) { ... }
```

Remove it. Rate limit responses are handled entirely by the limiter's `handler` callback and never reach the error handler middleware.

10. `taskSelect` should include `deletedAt` consistently

After adding soft deletes, `deletedAt` was added to `taskSelect` — but `updateTask` uses a separate inline select that may not include it:

typescript

```typescript
// task.service.ts updateTask — verify this select matches taskSelect
const task = await tx.task.update({
  where: { id: taskId },
  data,
  select: { ...taskSelect, createdBy: { select: { id: true, email: true } } },
});
```

`taskSelect` already includes `createdBy`, so the spread here creates a duplicate. Fix:

typescript

```typescript
select: taskSelect,
```

Summary of changes:

Issue	Fix
`userId` missing in controllers	Pass `req.user!.userId` consistently
Inline pagination parsing	Extract `validatePagination` to `src/lib/pagination.ts`
Broken `tx` type on `createNotification`	Use `Prisma.TransactionClient`
Inconsistent `tx` passing style	Params object everywhere
Raw role string in `deleteComment`	Use `can()` from `permissions.ts`
Duplicate `roleHierarchy`	Extract to `src/lib/roles.ts`
Stale VIEWER exclusion in validation	Confirm Zod schema is the live path
Unreachable 429 block in error handler	Remove it
Duplicate `createdBy` in `updateTask` select	Use `taskSelect` directly

None of these are breaking bugs in isolation, but together they create drift — the codebase starts having two ways to do the same thing, which makes onboarding harder and bugs easier to introduce. The highest priority fixes are the `roleHierarchy` duplication and the `can()` bypass in `deleteComment`, since those affect correctness of authorization logic.