

# TaskFlow Project Specification

## Project Overview

TaskFlow is a RESTful task management API built to demonstrate enterprise-grade backend architecture and team collaboration workflows. The system enables organizations to manage tasks across teams with role-based access control, ensuring proper separation of concerns between administrators and team members.

---

## Core Objectives

- Build a production-ready API with comprehensive authentication and authorization
  - Implement a hierarchical role system with distinct permissions
  - Demonstrate mastery of NestJS architectural patterns (pipes, guards, interceptors, DTOs)
  - Establish robust data persistence with relational database design
  - Create scalable endpoints with pagination and filtering capabilities
- 

## User Roles & Permissions

### Superadmin

- Full system access
- Can manage admin and user accounts
- Can view, create, update, and delete any task
- Can assign tasks to any user
- System configuration access

### Admin

- Can manage users within their scope
- Can create and assign tasks to users
- Can view all tasks within their domain
- Can update and delete tasks they created
- Cannot modify superadmin settings

## User

- Can register and manage their own account
  - Can create personal tasks
  - Can view tasks assigned to them
  - Can update and complete their own tasks
  - Can mark assigned tasks as complete
  - Cannot assign tasks to others
  - Cannot delete tasks assigned by admins
- 

## Functional Requirements

### Authentication & Authorization

- User registration with email validation
- Secure login with JWT token generation
- Password hashing and secure storage
- Token-based session management
- Role-based access control (RBAC) for all endpoints
- Refresh token mechanism for extended sessions

### Task Management

- Create tasks with title, description, priority, due date
- Assign tasks to specific users (admin/superadmin only)
- Update task details and status
- Mark tasks as complete
- Delete tasks (with permission checks)
- Task status workflow: Open → In Progress → Completed
- Priority levels: Low, Medium, High, Critical

### Data Operations

- Pagination for all list endpoints (page, limit parameters)

- Filtering by: status, priority, assignee, date range, creator
- Sorting by: creation date, due date, priority, status
- Search functionality for task titles and descriptions

## Validation & Error Handling

- DTO validation for all incoming requests
  - Standardized error responses
  - Input sanitization to prevent injection attacks
  - Meaningful validation messages
- 

## Technical Architecture

### Database Schema Design

#### Users Table

- `id` (UUID, primary key)
- `email` (unique, indexed)
- `password` (hashed)
- `firstName`
- `lastName`
- `role` (enum: 'user', 'admin', 'superadmin')
- `isActive` (boolean)
- `createdAt`
- `updatedAt`

#### Tasks Table

- `id` (UUID, primary key)
- `title`
- `description`
- `status` (enum: 'open', 'in\_progress', 'completed')

- `priority` (enum: 'low', 'medium', 'high', 'critical')
- `dueDate`
- `createdById` (foreign key → Users)
- `assignedToId` (foreign key → Users, nullable)
- `createdAt`
- `updatedAt`
- `completedAt` (nullable)

## Relationships

- One-to-Many: User (creator) → Tasks
  - One-to-Many: User (assignee) → Tasks
- 

## API Endpoints Structure

### Authentication

- `POST /auth/register` - Register new user
- `POST /auth/login` - User login
- `POST /auth/refresh` - Refresh access token
- `POST /auth/logout` - User logout

### Users

- `GET /users` - List all users (admin/superadmin only, with pagination)
- `GET /users/:id` - Get user details
- `PATCH /users/:id` - Update user
- `DELETE /users/:id` - Delete user (superadmin only)

### Tasks

- `GET /tasks` - List tasks (with pagination & filters)
- `GET /tasks/:id` - Get task details
- `POST /tasks` - Create new task

- `PATCH /tasks/:id` - Update task
  - `DELETE /tasks/:id` - Delete task
  - `POST /tasks/:id/assign` - Assign task to user (admin/superadmin only)
  - `PATCH /tasks/:id/complete` - Mark task as complete
- 

## NestJS Implementation Patterns

### Guards

- **JwtAuthGuard**: Validates JWT tokens on protected routes
- **RolesGuard**: Enforces role-based access control
- **TaskOwnershipGuard**: Ensures users can only modify their own tasks

### Pipes

- **ValidationPipe**: Global DTO validation
- **ParseUUIDPipe**: UUID parameter validation
- **ParseIntPipe**: Pagination parameter parsing

### Interceptors

- **TransformInterceptor**: Standardizes response format
- **LoggingInterceptor**: Request/response logging
- **TimeoutInterceptor**: Prevents long-running requests

### DTOs

- **Auth**: CreateUserDto, LoginDto, UpdateUserDto
- **Tasks**: CreateTaskDto, UpdateTaskDto, AssignTaskDto
- **Queries**: PaginationDto, TaskFilterDto

### Exception Filters

- Custom exception filter for consistent error responses
  - Validation exception formatting
-

# Data Validation Rules

## User Registration

- Email: valid format, unique
- Password: minimum 8 characters, requires uppercase, lowercase, number, special character
- First/Last Name: 2-50 characters

## Task Creation

- Title: required, 3-100 characters
  - Description: optional, max 500 characters
  - Priority: must be valid enum value
  - Due date: must be future date
- 

## Role-Based Access Control Matrix

Endpoint	User	Admin	Superadmin
POST /auth/register	✓	✓	✓
POST /auth/login	✓	✓	✓
GET /users	✗	✓	✓
GET /users/:id	Own only	✓	✓
PATCH /users/:id	Own only	✓	✓
DELETE /users/:id	✗	✗	✓
GET /tasks	Assigned/Own	All	All
POST /tasks	✓	✓	✓
PATCH /tasks/:id	Own only	Own/Created	✓
DELETE /tasks/:id	Own only	Own/Created	✓
POST /tasks/:id/assign	✗	✓	✓

---

## Success Criteria

### Functionality

- All CRUD operations working for users and tasks

- Role-based permissions properly enforced
- JWT authentication flow complete
- Pagination and filtering operational

## **Code Quality**

- Proper use of NestJS decorators and patterns
- Clean separation of concerns (controllers, services, repositories)
- Comprehensive DTO validation
- Error handling in all critical paths

## **Database**

- Proper foreign key relationships
- Indexes on frequently queried fields
- Migration files for schema management

## **Security**

- Passwords hashed with bcrypt
  - JWT secrets properly configured
  - SQL injection prevention through ORM
  - Input validation on all endpoints
- 

## **Development Phases**

### **Phase 1: Foundation**

- Database setup and entity definitions
- User authentication module
- Basic CRUD for users

### **Phase 2: Core Features**

- Task CRUD operations
- Role-based access control

- Task assignment functionality

### Phase 3: Enhancement

- Pagination and filtering
- Guards, pipes, and interceptors
- Comprehensive validation

### Phase 4: Polish

- Error handling refinement
- API documentation (Swagger/OpenAPI)
- Testing and bug fixes

---

## Environment Configuration

### Required Environment Variables

```
DATABASE_HOST
DATABASE_PORT
DATABASE_USER
DATABASE_PASSWORD
DATABASE_NAME
JWT_SECRET
JWT_EXPIRATION
REFRESH_TOKEN_SECRET
REFRESH_TOKEN_EXPIRATION
PORT
NODE_ENV
```

---

## Future Enhancement Opportunities

- Task comments and activity log
- File attachments for tasks
- Email notifications for task assignments
- Task dependencies and subtasks
- Team/project grouping

- Real-time updates via WebSockets
  - Task templates
  - Reporting and analytics dashboard
  - Task time tracking
  - Recurring tasks
- 

## **Testing Strategy**

### **Unit Tests**

- Service layer business logic
- Guard permission checks
- DTO validation rules

### **Integration Tests**

- API endpoint responses
- Database operations
- Authentication flow

### **E2E Tests**

- Complete user workflows
  - Role-based access scenarios
  - Error handling
- 

## **Success Metrics**

- All endpoints return appropriate HTTP status codes
- Unauthorized access properly blocked
- Data validation prevents invalid records
- Database queries optimized with proper relations
- API responses consistent and well-structured

- Authentication tokens expire and refresh correctly
  - Role hierarchy properly enforced
- 

## Technical Stack Summary

- **Framework:** NestJS
  - **Language:** TypeScript
  - **Database:** PostgreSQL
  - **ORM:** TypeORM
  - **Authentication:** JWT (jsonwebtoken)
  - **Validation:** class-validator, class-transformer
  - **Password Hashing:** bcrypt
- 

## Notes

### Role Column Design Decision

The system uses a single `role` enum column instead of multiple boolean columns (`isAdmin`, `isSuperAdmin`, etc.) for the following reasons:

- **Single source of truth:** No ambiguous states
- **Scalability:** Easy to add new roles without schema changes
- **Clean queries:** Simple WHERE clauses
- **Type safety:** Works perfectly with TypeScript enums
- **Industry standard:** Follows established RBAC patterns

### Why This Project

TaskFlow demonstrates production-ready backend development by incorporating:

- Authentication and authorization
- Complex business logic with role hierarchies
- Database relationships and queries
- Input validation and error handling

- RESTful API design principles
- Modern NestJS architectural patterns

This project serves as a comprehensive portfolio piece showcasing full-stack backend capabilities and enterprise development practices.

---

**Document Version:** 1.0

**Last Updated:** November 2025

**Project Status:** Specification Complete