**Issue Log API - Project Structure & Implementation Plan**

# API Design Overview

## Core Entities

* **Users**: Authentication and user management
* **Issues**: Main issue tracking entity
* **Comments**: Comments associated with issues
* **Files**: File attachments for issues

## RESTful API Endpoints

### Authentication

* POST /api/auth/register - User registration
* POST /api/auth/login - User login
* POST /api/auth/logout - User logout
* GET /api/auth/profile - Get user profile

### Issues

* GET /api/issues - List all issues (with pagination, filtering)
* GET /api/issues/:id - Get specific issue
* POST /api/issues - Create new issue
* PUT /api/issues/:id - Update issue
* DELETE /api/issues/:id - Delete issue
* PATCH /api/issues/:id/status - Update issue status (complete/pending)

### Comments

* GET /api/issues/:issueId/comments - Get comments for an issue
* POST /api/issues/:issueId/comments - Add comment to issue
* PUT /api/comments/:id - Update comment
* DELETE /api/comments/:id - Delete comment

### Files

* POST /api/issues/:issueId/files - Upload file to issue
* GET /api/issues/:issueId/files - Get files for an issue
* GET /api/files/:id/download - Download specific file
* DELETE /api/files/:id - Delete file

## Project Structure

https://github.com/dotasin/issue-log-app/blob/main/README.md

### Implementation Plan

**Phase 1: Project Setup & Core Infrastructure**

1. Initialize Node.js project with TypeScript
2. Set up database connection (MongoDB Atlas)
3. Configure logging and error handling
4. Set up authentication middleware

**Phase 2: Core API Development**

1. Implement User model and authentication
2. Create Issue CRUD operations
3. Add Comment functionality
4. Implement file upload/download

**Phase 3: Advanced Features & Polish**

1. Add pagination and filtering
2. Implement proper validation
3. Add comprehensive error handling
4. Create API documentation

**Technology Stack**

* **Runtime**: Node.js with TypeScript
* **Framework**: Express.js
* **Database**: MongoDB (Atlas)
* **Authentication**: JWT tokens
* **File Upload**: Multer
* **Validation**: Joi or express-validator
* **Logging**: Winston
* **Testing**: Jest (for future implementation)

**Key Assumptions**

1. **Authentication**: JWT-based authentication with refresh tokens
2. **File Storage**: Local file system storage (can be extended to cloud storage)
3. **Database**: MongoDB with Mongoose ODM
4. **File Size**: Maximum 10MB per file upload
5. **Pagination**: Default 20 items per page
6. **Issue Status**: Only "pending" and "complete" statuses
7. **User Roles**: Simple user system (can be extended with roles)

**Security Considerations**

* JWT token expiration and refresh mechanism
* Input validation on all endpoints
* File type restrictions for uploads
* Rate limiting (can be added with express-rate-limit)
* CORS configuration
* Environment variables for sensitive data

**Scalability & Maintenance**

* Modular architecture for easy extension
* Separation of concerns (controllers, services, models)
* Comprehensive error handling, Logging for monitoring and debugging, Type safety with TypeScript, Configuration through environment variables