partnr

Build Multi-Tenant SaaS Platform with Project & Task Management

Back

Mandatory Task

Domain

Backend Development

Frontend Development

Skills

API Development | API Documentation | Artificial Intelligence | Database Management | Email Security | Full-Stack Development | Management | Scalability | Solution Architecture | System Design | Authentication & Authorization

Difficulty

Hard

Tools

Docker | Express.Js | Git | JWT | Node.Js | PostgreSQL | Postman | React | Swagger | bcrypt

Pending Submission

Deadline: 27 Dec 2025, 04:59 pm. Please submit your work before the deadline.

TIME REMAINING

5d

Overview | Instructions | Resources | Submit

Description

Objective

Build a production-ready, multi-tenant SaaS application where multiple organizations (tenants) can independently register, manage their teams, create projects, and track tasks. The system must ensure complete data isolation between tenants, implement role-based access control (RBAC), and enforce subscription plan limits. This is a full-stack application requiring backend API development, frontend user interface, database design, a containerization.

Report Issue

Key Learning Outcomes:

partnr

- Implementing secure authentication and authorization
- Designing scalable database schemas with proper isolation
- Building RESTful APIs with proper error handling
- Creating responsive frontend applications
- Docker containerization and orchestration

Core Requirements

1. Multi-Tenancy Architecture

- **Data Isolation:** Each tenant's data must be completely isolated from other tenants. No tenant should be able to access another tenant's data, even through API manipulation.
- **Tenant Identification:** Every data record (except super_admin users) must be associated with a tenant via `tenant_id`.
- **Subdomain Support:** Each tenant gets a unique subdomain for identification during login.

2. Authentication & Authorization

- **JWT-Based Authentication:** Use JSON Web Tokens for stateless authentication with 24-hour expiry.
- **Three User Roles:**
  - **Super Admin:** System-level administrator with access to all tenants
  - **Tenant Admin:** Organization administrator with full control over their tenant
  - **User:** Regular team member with limited permissions
- **Role-Based Access Control:** Different API endpoints require different roles. Enforce authorization at the API level.

3. Database Schema Requirements

- **Core Tables:** tenants, users, projects, tasks, audit_logs (sessions table is optional - see details below)
- **Foreign Key Constraints:** Proper relationships with CASCADE delete where appropriate
- **Indexes:** Index on `tenant_id` columns for performance
- **Unique Constraints:** Email must be unique per tenant (not globally)
- **Super Admin Exception:** Super admin users have `tenant_id` as NULL

4. API Development Requirements

- **19 API Endpoints:** Covering authentication, tenant management, user management,

partnr

- **Proper HTTP Status Codes:** 200 (success), 201 (created), 400 (bad request), 401 (unauthorized), 403 (forbidden), 404 (not found), 409 (conflict)
- **Transaction Safety:** Critical operations (like tenant registration) must use database transactions
- **Audit Logging:** All important actions must be logged in `audit_logs` table

5. Subscription Management

- **Three Plans:** free, pro, enterprise
- **Plan Limits:** Each plan has `max_users` and `max_projects` limits
  - **Free Plan:** `max_users = 5`, `max_projects = 3`
  - **Pro Plan:** `max_users = 25`, `max_projects = 15`
  - **Enterprise Plan:** `max_users = 100`, `max_projects = 50`
- **Limit Enforcement:** APIs must check and enforce these limits before creating resources
- **Default Plan:** New tenants start with 'free' plan

6. Frontend Requirements

- **Six Main Pages:** Registration, Login, Dashboard, Projects List, Project Details, Users List
- **Protected Routes:** All pages except registration/login must require authentication
- **Role-Based UI:** Show/hide features based on user role
- **Responsive Design:** Must work on desktop and mobile devices
- **Error Handling:** Display user-friendly error messages

7. Docker Requirements (MANDATORY)

- **IMPORTANT:** Docker containerization is **MANDATORY** for submission. The application must be fully dockerized and runnable with `docker-compose up -d .`
- **All Three Services Required:** Database, Backend, and Frontend MUST all be containerized
- **Fixed Port Mappings (MANDATORY):**
  - Database: Port `5432` (external) → `5432` (internal)
  - Backend: Port `5000` (external) → `5000` (internal)
  - Frontend: Port `3000` (external) → `3000` (internal)
- **Fixed Service Names (MANDATORY):**
  - Database service: `database`
  - Backend service: `backend`
  - Frontend service: `frontend`
- **Docker Compose:** Complete `docker-compose.yml` with all three services
- **One-Command Deployment:** All services MUST start with single command: `docker-compose up -d`

- **Database Initialization (MANDATORY - Automatic Only):** Migrations and seed data MUST

connection status
- **Environment Variables:** All sensitive configuration must use environment variables. **All environment variables MUST be present in the repository** (either in `.env` file committed to repo, or directly in `docker-compose.yml`). Evaluation script needs access to all environment variables. Use test/development values only - do NOT use production secrets.
- **CORS Configuration:** Backend must be configured to accept requests from frontend (use `http://frontend:3000` in Docker network)
- **Inter-Service Communication:** Use service names (e.g., `http://backend:5000`) for communication, NOT `localhost`

8. Documentation Requirements

- **Research Document:** Multi-tenancy analysis, technology stack justification, security considerations
- **PRD:** User personas, functional requirements (15+), non-functional requirements (5+)
- **Architecture Document:** System architecture diagram, database ERD, API endpoint list
- **Technical Specification:** Project structure, setup guide
- **README.md:** Complete project documentation
- **API Documentation:** All 19 endpoints documented with examples

---

Implementation Details

Multi-Tenant SaaS Boilerplate - Complete Task Specification

PROJECT OVERVIEW

Project Title: Multi-Tenant SaaS Platform - Project & Task Management System

Objective: Build a production-ready, multi-tenant SaaS application where multiple organizations can register, manage their teams, create projects, and track tasks - all with proper data isolation, role-based access control, and subscription management.

STEP 1: RESEARCH & SYSTEM DESIGN

Step 1.1: Research & Requirements Analysis

Task 1.1.1: Research Document

partnr

Required Content:

1. **Multi-Tenancy Analysis**
   - Compare 3 multi-tenancy approaches:
     - Shared Database + Shared Schema (with tenant_id column)
     - Shared Database + Separate Schema (per tenant)
     - Separate Database (per tenant)
   - Create comparison table with pros/cons
   - Justify your chosen approach
   - Minimum 800 words
2. **Technology Stack Justification**
   - List chosen technologies for:
     - Backend framework
     - Frontend framework
     - Database
     - Authentication method
     - Deployment platforms
   - Explain WHY each technology was chosen
   - Mention alternatives considered
   - Minimum 500 words
3. **Security Considerations**
   - List 5 security measures for multi-tenant systems
   - Explain data isolation strategy
   - Authentication & authorization approach
   - Password hashing strategy
   - API security measures
   - Minimum 400 words

Task 1.1.2: Product Requirements Document (PRD)

File Location: `docs/PRD.md`

Required Content:

User Personas

partnr

- **Super Admin:** System-level administrator
- **Tenant Admin:** Organization administrator
- **End User:** Regular team member

For each persona, include:

- **Role description**
- **Key responsibilities**
- **Main goals**
- **Pain points they face**

Functional Requirements

- List a minimum of 15 functional requirements.
- Format: `"The system shall [requirement]"`
- **Examples:**
  - The system shall allow tenant registration with unique subdomain.
  - The system shall enforce subscription plan limits.
  - The system shall isolate tenant data completely.
- Organize by modules: Auth, Tenant, User, Project, Task
- Each requirement should be numbered (e.g., FR-001, FR-002, etc.)

Non-Functional Requirements

- Define at least 5 non-functional requirements.
- **Categories:**
  - **Performance:** e.g. API response time < 200ms for 90% of requests
  - **Security:** e.g. All passwords must be hashed, JWT expiry 24 hours
  - **Scalability:** e.g. Support minimum 100 concurrent users
  - **Availability:** e.g. 99% uptime target
  - **Usability:** e.g. Mobile responsive design
- Each requirement should be numbered (e.g., NFR-001, NFR-002, etc.)

Step 1.2: System Architecture Design

partnr

File Location: `docs/architecture.md`

Required Content:

1. **System Architecture Diagram**
   - High-level architecture diagram showing:
     - Client (Browser)
     - Frontend application
     - Backend API server
     - Database
     - Authentication flow
   - Save as image in `docs/images/system-architecture.png`
2. **Database Schema Design**
   - Entity Relationship Diagram (ERD)
   - Show all tables and relationships
   - Highlight foreign keys and indexes
   - Mark tenant_id columns for isolation
   - Save as `docs/images/database-erd.png`
3. **API Architecture**
   - List all API endpoints (minimum 15)
   - Organize by modules (Auth, Tenants, Users, Projects, Tasks)
   - Specify HTTP methods
   - Mark which endpoints require authentication
   - Mark which endpoints require specific roles

Task 1.2.2: Technical Specification

File Location: `docs/technical-spec.md`

Required Content:

1. **Project Structure**
   - Define complete folder structure for backend
   - Define complete folder structure for frontend
   - Explain purpose of each major folder

- Example:

partnr

```
|   ├── controllers/
|   ├── models/
|   ├── routes/
|   ├── middleware/
|   ├── utils/
|   └── config/
├── migrations/
└── tests/
```

2. **Development Setup Guide**

- Prerequisites (Node.js version, Python version, etc.)
- Environment variables needed
- Installation steps
- How to run locally
- How to run tests

STEP 2: DATABASE DESIGN & SETUP

Step 2.1: Database Schema Implementation

Task 2.1.1: Core Tables

File Location: `database/migrations/` or `backend/migrations/`

Required Tables (Must match exact schema names):

Table 1: tenants

Columns Required:

- `id` (Primary Key, VARCHAR/UUID)
- `name` (VARCHAR, NOT NULL)
- `subdomain` (VARCHAR, UNIQUE, NOT NULL)
- `status` (ENUM: 'active', 'suspended', 'trial')
- `subscription_plan` (ENUM: 'free', 'pro', 'enterprise')
- `max_users` (INTEGER, default based on plan)
- `max_projects` (INTEGER, default based on plan)
- `created_at` (TIMESTAMP)
- `updated_at` (TIMESTAMP)

Purpose: Store organization information

partnr

Table 2: users

Columns Required:

- `id` (Primary Key, VARCHAR/UUID)
- `tenant_id` (Foreign Key → tenants.id)
- `email` (VARCHAR, NOT NULL)
- `password_hash` (VARCHAR, NOT NULL)
- `full_name` (VARCHAR, NOT NULL)
- `role` (ENUM: 'super_admin', 'tenant_admin', 'user')
- `is_active` (BOOLEAN, DEFAULT true)
- `created_at` (TIMESTAMP)
- `updated_at` (TIMESTAMP)

Constraints:

- UNIQUE constraint on (tenant_id, email)
- Foreign key with CASCADE delete

Purpose: Store user accounts with tenant association

Table 3: projects

Columns Required:

- `id` (Primary Key, VARCHAR/UUID)
- `tenant_id` (Foreign Key → tenants.id)
- `name` (VARCHAR, NOT NULL)
- `description` (TEXT)
- `status` (ENUM: 'active', 'archived', 'completed')
- `created_by` (Foreign Key → users.id)
- `created_at` (TIMESTAMP)
- `updated_at` (TIMESTAMP)

Constraints:

- Foreign keys with CASCADE delete
- Index on tenant_id

Purpose: Store projects for each tenant

Table 4: tasks

partnr

- `id` (Primary Key, VARCHAR/UUID)
- `project_id` (Foreign Key → projects.id)
- `tenant_id` (Foreign Key → tenants.id)
- `title` (VARCHAR, NOT NULL)
- `description` (TEXT)
- `status` (ENUM: 'todo', 'in_progress', 'completed')
- `priority` (ENUM: 'low', 'medium', 'high')
- `assigned_to` (Foreign Key → users.id, NULLABLE)
- `due_date` (DATE, NULLABLE)
- `created_at` (TIMESTAMP)
- `updated_at` (TIMESTAMP)

Constraints:

- Foreign keys with CASCADE delete
- Index on (tenant_id, project_id)

Purpose: Store tasks within projects

Table 5: audit_logs

Columns Required:

- `id` (Primary Key, VARCHAR/UUID)
- `tenant_id` (Foreign Key → tenants.id)
- `user_id` (Foreign Key → users.id, NULLABLE)
- `action` (VARCHAR, NOT NULL) - e.g., 'CREATE_USER', 'DELETE_PROJECT'
- `entity_type` (VARCHAR) - e.g., 'user', 'project', 'task'
- `entity_id` (VARCHAR)
- `ip_address` (VARCHAR, NULLABLE)
- `created_at` (TIMESTAMP)

Purpose: Track all important actions for security audit

Table 6: sessions (OPTIONAL)

NOTE: This table is **OPTIONAL**. If you use JWT-only authentication, you can skip creating this table entirely.

If you choose to implement sessions:

partnr

- `user_id` (Foreign Key → users.id)
- `token` (VARCHAR, UNIQUE, NOT NULL)
- `expires_at` (TIMESTAMP, NOT NULL)
- `created_at` (TIMESTAMP)

- **Constraints:**
  - Foreign key with CASCADE delete
  - Index on token
- **Purpose:** Track active user sessions

If using JWT-only (recommended):

- You do NOT need to create this table
- JWT tokens are stateless and don't require session storage
- Simply skip this table in your migrations

Step 2.2: Database Migrations & Seeds

Task 2.2.1: Migration Files

File Location: `database/migrations/` or `backend/migrations/`

Requirements:

- Separate migration file for each table
- Naming convention: `001_create_tenants.sql` , `002_create_users.sql` , etc.
- Each file should have clear UP and DOWN migrations
- Migrations should run in order

Task 2.2.2: Seed Data

File Location: `database/seeds/seed_data.sql` or `backend/seeds/`

Required Seed Data:

1. **Super Admin Account:**
   - Email: `superadmin@system.com`
   - Password: `Admin@123` (hashed)
   - Role: `super_admin`

- Not associated with any tenant

partnr

- Name: `Demo Company`
- Subdomain: `demo`
- Status: `active`
- Plan: `pro`

3. **Tenant Admin for Demo Company:**
   - Email: `admin@demo.com`
   - Password: `Demo@123` (hashed)
   - Role: `tenant_admin`
4. **2 Regular Users for Demo Company:**
   - User 1: `user1@demo.com` / `User@123`
   - User 2: `user2@demo.com` / `User@123`
   - Role: `user`
5. **2 Sample Projects for Demo Company**
6. **5 Sample Tasks distributed across projects**

STEP 3: BACKEND API DEVELOPMENT

Code Structure Recommendations:

- **Authentication Middleware:** Create middleware to extract and validate JWT tokens, extract `tenant_id` and `role` from token payload
- **Authorization Middleware:** Create middleware to check user roles and tenant access permissions
- **Tenant Isolation Middleware:** Automatically filter queries by `tenant_id` from JWT (except for super_admin)
- **Audit Logging Service:** Create a utility/service function to log actions to `audit_logs` table
- **Error Handling:** Implement consistent error handling middleware for all API endpoints
- **Input Validation:** Use validation middleware or library (e.g., express-validator, joi) for request body validation

---

Step 3.1: Authentication Module

API Endpoints Required:

API 1: Tenant Registration

partnr

- **Authentication:** None (public)
- **Request Body Fields:**
  - `tenantName` (string, required)
  - `subdomain` (string, required, unique)
  - `adminEmail` (string, required, email format)
  - `adminPassword` (string, required, min 8 chars)
  - `adminFullName` (string, required)
- **Success Response (201):**

```
{
  "success": true,
  "message": "Tenant registered successfully",
  "data": {
    "tenantId": "uuid",
    "subdomain": "value",
    "adminUser": {
      "id": "uuid",
      "email": "value",
      "fullName": "value",
      "role": "tenant_admin"
    }
  }
}
```

- **Error Responses:**
  - 400: Validation errors
  - 409: Subdomain or email already exists
- **Business Logic Requirements:**
  - Hash password using bcrypt/argon2
  - Create tenant record
  - Create admin user with role `tenant_admin`
  - Both operations in single database transaction
  - Set default subscription limits based on 'free' plan

Test Inputs for Evaluation:

```
{
  "tenantName": "Test Company Alpha",
  "subdomain": "testalpha",
```

```
  "adminEmail": "admin@testalpha.com",
```

partnr

API 2: User Login

- **Endpoint:** `POST /api/auth/login`
- **Authentication:** None (public)
- **Request Body Fields:**
  - `email` (string, required)
  - `password` (string, required)
  - `tenantSubdomain` (string, required) OR `tenantId` (string)
- **Success Response (200):**

```
{
  "success": true,
  "data": {
    "user": {
      "id": "uuid",
      "email": "value",
      "fullName": "value",
      "role": "tenant_admin",
      "tenantId": "uuid"
    },
    "token": "jwt-token-string",
    "expiresIn": 86400
  }
}
```

- **Error Responses:**
  - 401: Invalid credentials
  - 404: Tenant not found
  - 403: Account suspended/inactive
- **Business Logic Requirements:**
  - Verify tenant exists and is active
  - Verify user belongs to that tenant
  - Verify password hash matches
  - Generate JWT token containing: `{userId, tenantId, role}`
  - Token expiry: 24 hours

- Optional: Create session record

partnr

```
{
  "email": "admin@demo.com",
  "password": "Demo@123",
  "tenantSubdomain": "demo"
}
```

API 3: Get Current User

- **Endpoint:** `GET /api/auth/me`
- **Authentication:** Required (JWT token)
- **Headers:** `Authorization: Bearer {token}`
- **Success Response (200):**

```
{
  "success": true,
  "data": {
    "id": "uuid",
    "email": "value",
    "fullName": "value",
    "role": "tenant_admin",
    "isActive": true,
    "tenant": {
      "id": "uuid",
      "name": "value",
      "subdomain": "value",
      "subscriptionPlan": "pro",
      "maxUsers": 10,
      "maxProjects": 20
    }
  }
}
```

- **Error Responses:**
  - 401: Token invalid/expired/missing
  - 404: User not found
- **Business Logic Requirements:**
  - Verify JWT token
  - Extract userId from token

- Join with tenants table to get tenant info

partnr

API 4: Logout

- **Endpoint:** `POST /api/auth/logout`
- **Authentication:** Required
- **Headers:** `Authorization: Bearer {token}`
- **Success Response (200):**

```
{
  "success": true,
  "message": "Logged out successfully"
}
```

- **Business Logic Requirements:**
  - If using session table: Delete session record
  - If JWT only: Return success (client removes token)
  - Log action in audit_logs

Step 3.2: Tenant Management Module

API Endpoints Required:

API 5: Get Tenant Details

- **Endpoint:** `GET /api/tenants/:tenantId`
- **Authentication:** Required
- **Authorization:** User must belong to this tenant OR be super_admin
- **Success Response (200):**

```
{
  "success": true,
  "data": {
    "id": "uuid",
    "name": "value",
    "subdomain": "value",
    "status": "active",
    "subscriptionPlan": "pro",
```

```
    "maxUsers": 10,
```

partnr

```
      "totalUsers": 5,
      "totalProjects": 3,
      "totalTasks": 15
    }
  }
}
```

- **Error Responses:**
  - 403: Unauthorized access
  - 404: Tenant not found
- **Business Logic Requirements:**
  - Verify user belongs to tenant OR is super_admin
  - Calculate stats (count from related tables)

Test Scenario:

- Login as demo tenant admin
- Access own tenant details: Should succeed
- Access different tenant details: Should fail with 403

API 6: Update Tenant

- **Endpoint:** `PUT /api/tenants/:tenantId`
- **Authentication:** Required
- **Authorization:** tenant_admin OR super_admin only
- **Request Body Fields:**
  - `name` (string, optional)
  - `status` (enum, optional) - super_admin only
  - `subscriptionPlan` (enum, optional) - super_admin only
  - `maxUsers` (integer, optional) - super_admin only
  - `maxProjects` (integer, optional) - super_admin only
- **Success Response (200):**

```
{
  "success": true,
  "message": "Tenant updated successfully",
```

```
    "data": {
```

partnr

```
    }
  }
```

- **Business Logic Requirements:**
  - Tenant admins can only update `name`
  - Super admins can update all fields
  - Log changes in audit_logs table
  - Return 403 if tenant_admin tries to update restricted fields

Test Inputs:

```
{
  "name": "Updated Company Name"
}
```

API 7: List All Tenants

- **Endpoint:** `GET /api/tenants`
- **Authentication:** Required
- **Authorization:** super_admin ONLY
- **Query Parameters:**
  - `page` (integer, default: 1)
  - `limit` (integer, default: 10, max: 100)
  - `status` (enum, optional filter)
  - `subscriptionPlan` (enum, optional filter)
- **Success Response (200):**

```
{
  "success": true,
  "data": {
    "tenants": [
      {
        "id": "uuid",
        "name": "value",
        "subdomain": "value",
        "status": "active",
        "subscriptionPlan": "pro",
```

```
        "totalUsers": 5,
```

partnr

```
      ],
      "pagination": {
        "currentPage": 1,
        "totalPages": 5,
        "totalTenants": 47,
        "limit": 10
      }
    }
  }
```

- **Error Responses:**
  - 403: Not super_admin
- **Business Logic Requirements:**
  - Return 403 if user role is not super_admin
  - Implement pagination
  - Calculate totalUsers and totalProjects for each tenant
  - Support filtering by status and plan

Test Scenario:

- Login as super_admin: Should see all tenants
- Login as tenant_admin: Should get 403

Step 3.3: User Management Module

API Endpoints Required:

API 8: Add User to Tenant

- **Endpoint:** `POST /api/tenants/:tenantId/users`
- **Authentication:** Required
- **Authorization:** tenant_admin only
- **Request Body Fields:**
  - `email` (string, required)
  - `password` (string, required, min 8 chars)
  - `fullName` (string, required)
  - `role` (enum: 'user' or 'tenant_admin', default: 'user')

- **Success Response (201):**

partnr

```
{
  "success": true,
  "message": "User created successfully",
  "data": {
    "id": "uuid",
    "email": "value",
    "fullName": "value",
    "role": "user",
    "tenantId": "uuid",
    "isActive": true,
    "createdAt": "timestamp"
  }
}
```

- **Error Responses:**
  - 403: Subscription limit reached OR not authorized
  - 409: Email already exists in this tenant
- **Business Logic Requirements:**
  - Check current user count vs maxUsers from tenant
  - Return 403 if limit reached
  - Hash password
  - Email must be unique per tenant (can exist in different tenants)
  - Log in audit_logs

Test Inputs:

```
{
  "email": "newuser@demo.com",
  "password": "NewUser@123",
  "fullName": "New User",
  "role": "user"
}
```

API 9: List Tenant Users

- **Endpoint:** `GET /api/tenants/:tenantId/users`
- **Authentication:** Required
- **Authorization:** User must belong to this tenant
- **Query Parameters:**

- `search` (string, optional) - Search by name or email

partnr

- `limit` (integer, optional, default: 50, max: 100) - Items per page

- **Success Response (200):**

```
{
  "success": true,
  "data": {
    "users": [
      {
        "id": "uuid",
        "email": "value",
        "fullName": "value",
        "role": "tenant_admin",
        "isActive": true,
        "createdAt": "timestamp"
      }
    ],
    "total": 5,
    "pagination": {
      "currentPage": 1,
      "totalPages": 1,
      "limit": 50
    }
  }
}
```

- **Business Logic Requirements:**
  - Filter by tenantId automatically
  - Do NOT return password_hash
  - Order by createdAt DESC
  - Support search by name or email (case-insensitive)
  - Support filtering by role
  - Support pagination (optional but recommended for large datasets)

API 10: Update User

- **Endpoint:** `PUT /api/users/:userId`
- **Authentication:** Required
- **Authorization:** tenant_admin OR self (limited fields)
- **Request Body Fields:**

- `fullName` (string, optional)

partnr

- **Success Response (200):**

```
{
  "success": true,
  "message": "User updated successfully",
  "data": {
    "id": "uuid",
    "fullName": "updated-value",
    "role": "user",
    "updatedAt": "timestamp"
  }
}
```

- **Business Logic Requirements:**
  - Users can update their own fullName
  - Only tenant_admin can update role and isActive
  - Verify user belongs to same tenant
  - Log in audit_logs

API 11: Delete User

- **Endpoint:** `DELETE /api/users/:userId`
- **Authentication:** Required
- **Authorization:** tenant_admin only
- **Success Response (200):**

```
{
  "success": true,
  "message": "User deleted successfully"
}
```

- **Error Responses:**
  - 403: Cannot delete self OR not authorized
  - 404: User not found
- **Business Logic Requirements:**
  - tenant_admin cannot delete themselves

- Verify user belongs to same tenant

partnr

Step 3.4: Project Management Module

API Endpoints Required:

API 12: Create Project

- **Endpoint:** `POST /api/projects`
- **Authentication:** Required
- **Request Body Fields:**
  - `name` (string, required)
  - `description` (text, optional)
  - `status` (enum, optional, default: 'active')
- **Success Response (201):**

```
{
  "success": true,
  "data": {
    "id": "uuid",
    "tenantId": "uuid",
    "name": "value",
    "description": "value",
    "status": "active",
    "createdBy": "uuid",
    "createdAt": "timestamp"
  }
}
```

- **Error Responses:**
  - 403: Project limit reached
- **Business Logic Requirements:**
  - Get tenantId from JWT token automatically
  - Get createdBy from JWT token automatically
  - Check current project count vs maxProjects
  - Return 403 if limit reached

Test Inputs:

partnr

```
}
```

API 13: List Projects

- **Endpoint:** `GET /api/projects`
- **Authentication:** Required
- **Query Parameters:**
  - `status` (enum, optional filter) - Filter by project status
  - `search` (string, optional) - Search by project name (case-insensitive)
  - `page` (integer, optional, default: 1) - For pagination
  - `limit` (integer, optional, default: 20, max: 100) - Items per page
- **Success Response (200):**

```
{
  "success": true,
  "data": {
    "projects": [
      {
        "id": "uuid",
        "name": "value",
        "description": "value",
        "status": "active",
        "createdBy": {
          "id": "uuid",
          "fullName": "value"
        },
        "taskCount": 5,
        "completedTaskCount": 2,
        "createdAt": "timestamp"
      }
    ],
    "total": 3,
    "pagination": {
      "currentPage": 1,
      "totalPages": 1,
      "limit": 20
    }
  }
}
```

- **Business Logic Requirements:**

- Filter by user's tenantId automatically

partnr

- Support status filtering
- Support search by name (case-insensitive)
- Support pagination (optional but recommended for large datasets)

API 14: Update Project

- **Endpoint:** `PUT /api/projects/:projectId`
- **Authentication:** Required
- **Authorization:** tenant_admin OR project creator
- **Request Body Fields:**
  - `name` (string, optional)
  - `description` (text, optional)
  - `status` (enum: 'active'/'archived'/'completed', optional)
- **Success Response (200):**

```
{
  "success": true,
  "message": "Project updated successfully",
  "data": {
    "id": "uuid",
    "name": "updated-value",
    "description": "updated-value",
    "status": "active",
    "updatedAt": "timestamp"
  }
}
```

- **Error Responses:**
  - 403: Not authorized (user doesn't belong to tenant OR not creator/admin)
  - 404: Project not found OR belongs to different tenant
- **Business Logic Requirements:**
  - Verify project belongs to user's tenant
  - Only tenant_admin or createdBy user can update
  - Update only provided fields (partial update)
  - Log in audit_logs

Test Inputs:

partnr

```
  "name": "Updated Project Name",
  "description": "Updated description",
  "status": "archived"
}
```

API 15: Delete Project

- **Endpoint:** `DELETE /api/projects/:projectId`
- **Authentication:** Required
- **Authorization:** tenant_admin OR project creator
- **Success Response (200):**

```
{
  "success": true,
  "message": "Project deleted successfully"
}
```

- **Error Responses:**
  - 403: Not authorized
  - 404: Project not found OR belongs to different tenant
- **Business Logic Requirements:**
  - Verify project belongs to user's tenant
  - Only tenant_admin or createdBy user can delete
  - Cascade delete tasks OR handle foreign key constraint
  - Log in audit_logs

Step 3.5: Task Management Module

API Endpoints Required:

API 16: Create Task

- **Endpoint:** `POST /api/projects/:projectId/tasks`
- **Authentication:** Required

- **Request Body Fields:**

partnr

  - `description` (text, optional)
  - `assignedTo` (uuid, optional)
  - `priority` (enum: 'low'/'medium'/'high', default: 'medium')
  - `dueDate` (date, optional)

- **Success Response (201):**

```
{
  "success": true,
  "data": {
    "id": "uuid",
    "projectId": "uuid",
    "tenantId": "uuid",
    "title": "value",
    "description": "value",
    "status": "todo",
    "priority": "high",
    "assignedTo": "uuid",
    "dueDate": "2024-07-01",
    "createdAt": "timestamp"
  }
}
```

- **Error Responses:**
  - 403: Project doesn't belong to user's tenant
  - 400: assignedTo user doesn't belong to same tenant
- **Business Logic Requirements:**
  - Verify project exists and belongs to user's tenant
  - Get tenantId from project (not from JWT)
  - If assignedTo provided, verify user belongs to same tenant
  - Default status: 'todo'

Test Inputs:

```
{
  "title": "Design homepage mockup",
  "description": "Create high-fidelity design",
  "assignedTo": "user-uuid-here",
  "priority": "high",
  "dueDate": "2024-07-15"
}
```

API 17: List Project Tasks

partnr

- **Authentication:** Required
- **Query Parameters:**
  - `status` (enum, optional filter) - Filter by task status
  - `assignedTo` (uuid, optional filter) - Filter by assigned user
  - `priority` (enum, optional filter) - Filter by priority
  - `search` (string, optional) - Search by task title (case-insensitive)
  - `page` (integer, optional, default: 1) - For pagination
  - `limit` (integer, optional, default: 50, max: 100) - Items per page
- **Success Response (200):**

```
{
  "success": true,
  "data": {
    "tasks": [
      {
        "id": "uuid",
        "title": "value",
        "description": "value",
        "status": "in_progress",
        "priority": "high",
        "assignedTo": {
          "id": "uuid",
          "fullName": "value",
          "email": "value"
        },
        "dueDate": "2024-07-01",
        "createdAt": "timestamp"
      }
    ],
    "total": 5,
    "pagination": {
      "currentPage": 1,
      "totalPages": 1,
      "limit": 50
    }
  }
}
```

- **Business Logic Requirements:**
  - Verify project belongs to user's tenant
  - Join with users table for assignedTo details
  - Support all query parameter filters

- Support search by title (case-insensitive)

partnr

API 18: Update Task Status

- **Endpoint:** `PATCH /api/tasks/:taskId/status`
- **Authentication:** Required
- **Request Body Fields:**
  - `status` (enum: 'todo'/'in_progress'/'completed', required)
- **Success Response (200):**

```
{
  "success": true,
  "data": {
    "id": "uuid",
    "status": "completed",
    "updatedAt": "timestamp"
  }
}
```

- **Business Logic Requirements:**
  - Verify task belongs to user's tenant
  - Any user in tenant can update status
  - Update only status field

API 19: Update Task

- **Endpoint:** `PUT /api/tasks/:taskId`
- **Authentication:** Required
- **Request Body Fields:**
  - `title` (string, optional)
  - `description` (text, optional)
  - `status` (enum: 'todo'/'in_progress'/'completed', optional)
  - `priority` (enum: 'low'/'medium'/'high', optional)
  - `assignedTo` (uuid, optional, can be null to unassign)
  - `dueDate` (date, optional, can be null)

- **Success Response (200):**

partnr

```
{
  "success": true,
  "message": "Task updated successfully",
  "data": {
    "id": "uuid",
    "title": "updated-value",
    "description": "updated-value",
    "status": "in_progress",
    "priority": "high",
    "assignedTo": {
      "id": "uuid",
      "fullName": "value",
      "email": "value"
    },
    "dueDate": "2024-07-20",
    "updatedAt": "timestamp"
  }
}
```

- **Error Responses:**
  - 403: Task doesn't belong to user's tenant
  - 400: assignedTo user doesn't belong to same tenant
  - 404: Task not found
- **Business Logic Requirements:**
  - Verify task belongs to user's tenant
  - If assignedTo provided, verify user belongs to same tenant
  - Update only provided fields (partial update)
  - If assignedTo is null, unassign the task
  - Log in audit_logs

Test Inputs:

```
{
  "title": "Updated task title",
  "description": "Updated description",
  "priority": "high",
  "assignedTo": "user-uuid-here",
  "dueDate": "2024-08-01"
}
```

STEP 4: FRONTEND DEVELOPMENT

Step 4.1: Authentication Pages

partnr

Route: `/register`

Required Elements:

- Form with fields:
  - Organization Name (input)
  - Subdomain (input with preview: subdomain.yourapp.com)
  - Admin Email (email input)
  - Admin Full Name (input)
  - Password (password input with show/hide)
  - Confirm Password (password input)
  - Terms & Conditions checkbox
- Form validation (client-side)
- Submit button with loading state
- Link to login page
- Error message display area
- Success message with redirect to login

API Integration:

- Call `POST /api/auth/register-tenant`
- Handle success: Show message, redirect to login
- Handle errors: Display validation errors

Page 2: Login Page

Route: `/login`

Required Elements:

- Form with fields:
  - Email (email input)
  - Password (password input)
  - Tenant Subdomain (input)
- Remember me checkbox (optional)
- Submit button with loading state
- Link to register page
- Error message display
- Success: Store token, redirect to dashboard

API Integration:

partnr

- Store JWT token in localStorage/sessionStorage
- Redirect to `/dashboard` on success

Protected Route Implementation

Requirements:

- Create authentication middleware/guard
- Check for valid token
- Redirect to login if not authenticated
- Verify token on app load
- Auto-logout on token expiry

Step 4.2: Dashboard & Navigation

Component 1: Navigation Bar

Required Elements:

- Logo/App name
- Navigation menu:
  - Dashboard
  - Projects
  - Tasks (if tenant_admin/super_admin)
  - Users (if tenant_admin)
  - Tenants (if super_admin only)
- User dropdown menu:
  - Profile
  - Settings
  - Logout
- Display current user name and role
- Responsive design (hamburger menu on mobile)

Page 3: Dashboard Page

Route: `/dashboard`

Required Elements:

Statistics Cards (Top Section):

partnr

- Total Tasks count
- Completed Tasks count
- Pending Tasks count

Recent Projects Section:

- List of 5 most recent projects
- Each showing: name, status, task count
- Click to navigate to project details

My Tasks Section:

- List of tasks assigned to current user
- Filter by status
- Shows: title, project name, priority, due date

API Integrations Needed:

- `GET /api/auth/me` - Get current user and tenant info
- `GET /api/projects` - Get projects list
- `GET /api/projects/:id/tasks?assignedTo=currentUserId` - Get user's tasks

Step 4.3: Project & Task Management

Page 4: Projects List Page

Route: `/projects`

Required Elements:

- "Create New Project" button
- Projects displayed in cards/table
- Each project shows:
  - Name
  - Description (truncated)
  - Status badge
  - Task count
  - Created date
  - Creator name
  - Actions: View, Edit, Delete
- Filter by status dropdown

- Search by name

partnr

API Integration:

- `GET /api/projects?status=filter`
- `DELETE /api/projects/:id` (with confirmation)

Component 2: Create/Edit Project Modal

Required Elements:

- Modal/dialog for create/edit
- Form fields:
  - Project Name (input, required)
  - Description (textarea)
  - Status (dropdown: active/archived/completed)
- Cancel and Save buttons
- Form validation

API Integration:

- `POST /api/projects` for create
- `PUT /api/projects/:id` for edit

Page 5: Project Details Page

Route: `/projects/:projectId`

Required Elements:

Project Header:

- Project name (editable inline)
- Status badge
- Description
- Edit and Delete buttons

Tasks Section:

- "Add Task" button
- Task list in table/cards format
- Each task shows:
  - Title

- Status badge

partnr

- Due date
- Actions: Edit, Change Status, Delete

- Filter by status, priority, assigned user
- Group by status (Todo, In Progress, Completed) - Optional Kanban style

API Integration:

- `GET /api/projects/:id` - Project details
- `PUT /api/projects/:id` - Update project
- `GET /api/projects/:id/tasks` - Tasks list
- `POST /api/projects/:id/tasks` - Create task
- `PUT /api/tasks/:id` - Update task (all fields)
- `PATCH /api/tasks/:id/status` - Update task status only
- `DELETE /api/tasks/:id` - Delete task

Step 4.4: User Management

Page 6: Users List Page

(Only visible to tenant_admin)

Route: `/users`

Required Elements:

- "Add User" button
- Users displayed in table
- Columns:
  - Full Name
  - Email
  - Role badge
  - Status (Active/Inactive)
  - Created Date
  - Actions: Edit, Delete
- Search by name/email
- Filter by role

API Integration:

- `GET /api/tenants/:tenantId/users`

- `DELETE /api/users/:id`

partnr

Component 3: Add/Edit User Modal

Required Elements:

- Modal for add/edit user
- Form fields:
  - Email (input, required)
  - Full Name (input, required)
  - Password (input, required for add, optional for edit)
  - Role (dropdown: user/tenant_admin)
  - Active Status (checkbox)
- Form validation
- Cancel and Save buttons

API Integration:

- `POST /api/tenants/:tenantId/users`
- `PUT /api/users/:id`

STEP 5: DEVOPS & DEPLOYMENT

MANDATORY REQUIREMENTS:

- **Docker Configuration:** Docker setup with `docker-compose.yml` and `Dockerfile(s)` is **MANDATORY** for production-ready status
- **Production Deployment:** Both frontend and backend must be deployed to production platforms and accessible via HTTPS
- **Health Check:** Health check endpoint must be functional on deployed backend

---

Step 5.1: Environment Configuration

Task 5.1.1: Environment Variables Setup

File Location: `.env.example` in backend root

Required Variables:

```
#### Database
DB_HOST=localhost
DB_PORT=5432
DB_NAME=saas_db
```

```
DB_USER=postgres
```

partnr

```
JWT_SECRET=your_jwt_secret_key_min_32_chars
JWT_EXPIRES_IN=24h

#### Server
PORT=5000
NODE_ENV=development

#### Frontend URL (for CORS)
FRONTEND_URL=http://localhost:3000

#### Optional: Email configuration
EMAIL_HOST=
EMAIL_PORT=
EMAIL_USER=
EMAIL_PASSWORD=
```

Requirements:

- `.env.example` with all variables documented (template with placeholder values)
- **For Evaluation:** `.env` file MUST be committed to repository (NOT in .gitignore) with test/development values
- **Alternative:** Environment variables can be directly in `docker-compose.yml` instead of `.env` file
- Backend reads from environment variables
- **Important:** Evaluation script needs access to all environment variables. Use test/development values only - do NOT use production secrets.

CORS Configuration:

- Configure CORS middleware to allow requests from `FRONTEND_URL` environment variable
- **For Docker setup:** Set `FRONTEND_URL=http://frontend:3000` (use service name, NOT localhost)
- **For local development:** Set `FRONTEND_URL=http://localhost:3000`
- Example configuration:

```
// Allow requests from frontend URL
// In Docker: http://frontend:3000
// In local dev: http://localhost:3000
app.use(cors({
  origin: process.env.FRONTEND_URL || 'http://localhost:3000',
  credentials: true
}));
```

- **Important:** In Docker network, use service names (e.g., `http://frontend:3000`) for inter-service communication
- Ensure CORS is configured before API routes

partnr

Task 5.2.1: Docker Setup

IMPORTANT: Docker configuration is **MANDATORY** for evaluation. Your application must be fully containerized with a working `docker-compose.yml` file and `Dockerfile(s)`. The application must be fully functional when started with `docker-compose up -d`.

File Location: `docker-compose.yml` in **repository root** (same level as README.md)

Required Services (ALL THREE MANDATORY):

- **Database (PostgreSQL)** - MUST be named `database`
- **Backend API** - MUST be named `backend`
- **Frontend** - MUST be named `frontend` and MUST be containerized

IMPORTANT: All three services MUST be defined in docker-compose.yml and MUST start with a single command: `docker-compose up -d`

Fixed Port Mappings (MANDATORY):

- **Database:** External port `5432` → Internal port `5432`
- **Backend:** External port `5000` → Internal port `5000`
- **Frontend:** External port `3000` → Internal port `3000`

Sample Structure:

```
version: '3.8'
services:
  database:
    image: postgres:15
    container_name: database
    environment:
      POSTGRES_DB: saas_db
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
    ports:
      - "5432:5432"  # Fixed: external:internal
    volumes:
      - db_data:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres"]
      interval: 10s
      timeout: 5s
      retries: 5

  backend:
    build: ./backend
    container_name: backend
    ports:
```

```
      - "5000:5000"  # Fixed: external:internal
```

partnr

```
      DB_NAME: saas_db
      DB_USER: postgres
      DB_PASSWORD: postgres
      JWT_SECRET: your-secret-key
      FRONTEND_URL: http://frontend:3000  # Use service name, not localhost
    depends_on:
      database:
        condition: service_healthy
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:5000/api/health"]  # localhost is correct here
      interval: 10s
      timeout: 5s
      retries: 5
      # Note: If curl is not available, use wget or create a custom health check script

  frontend:
    build: ./frontend
    container_name: frontend
    ports:
      - "3000:3000"  # Fixed: external:internal
    environment:
      REACT_APP_API_URL: http://backend:5000/api  # Use service name, not localhost
    depends_on:
      backend:
        condition: service_healthy

volumes:
  db_data:
```

Note:

- Use service names (e.g., `database`, `backend`, `frontend`) for inter-service communication, NOT `localhost`
- All ports are fixed and MUST match the specification above

Requirements:

- `docker-compose.yml` file present in **repository root** (same level as README.md)
- `Dockerfile` for backend (MANDATORY) - location: `./backend/Dockerfile` or as specified in docker-compose.yml
- `Dockerfile` for frontend (MANDATORY) - location: `./frontend/Dockerfile` or as specified in docker-compose.yml
- Database service MUST be named `database`
- Backend service MUST be named `backend`
- Frontend service MUST be named `frontend`
- All three services MUST use fixed port mappings (5432, 5000, 3000)

- All services start successfully with single command: `docker-compose up -d`

partnr

Frontend Build Approach (Your Choice): You can choose either approach - both are acceptable:

- **Production Build:** Build static files during `docker build`, serve with nginx/serve/static file server
- **Development Server:** Run dev server (e.g., `npm start`, `yarn dev`) in container
- **Framework Production Server:** Use framework's production server (e.g., Next.js production mode, Vue production build)

The evaluation script only cares that frontend is accessible at `http://localhost:3000` after `docker-compose up -d`. Choose the approach that works best for your framework.

Health Check Implementation: The health check example uses `curl`, but you can use:

- `curl` (if available in container)
- `wget` (if available)
- Custom health check script
- Framework-specific health check endpoint
- Or omit health check if your framework provides built-in health endpoints

The health check should verify the service is ready to accept requests. If your backend doesn't have curl/wget, create a simple health check script or use your framework's built-in health check mechanism.

Verification Checklist: After running `docker-compose up -d`, verify:

- [ ] All 3 services show "Up" status: `docker-compose ps`
- [ ] Health check responds: `curl http://localhost:5000/api/health` (returns 200)
- [ ] Frontend accessible: Open `http://localhost:3000` in browser
- [ ] Can login with credentials from `submission.json`
- [ ] Database contains seed data (test login to verify)

Task 5.2.2: Database Initialization (MANDATORY)

CRITICAL: Your Docker setup MUST automatically initialize the database with migrations and seed data for automated evaluation.

MANDATORY - Automatic Initialization Only:

- Database migrations MUST run automatically when backend service starts
- Seed data MUST load automatically after migrations complete

- **NO manual commands allowed** - everything must be automatic

partnr

  - Init container in docker-compose.yml
  - Backend startup script that runs migrations/seeds before starting the server
- Evaluation script will only run `docker-compose up -d` - all initialization must happen automatically

Seed Data Requirements (Flexible but Must Be Documented):

You can choose your own seed data, but MUST meet these minimum requirements:

1. **Super Admin (REQUIRED):**
   - At least one user with role `super_admin`
   - `tenant_id` MUST be `NULL`
   - Password MUST be properly hashed with bcrypt/argon2
2. **Tenants (REQUIRED):**
   - At least one tenant with status `active`
   - Each tenant MUST have a unique subdomain
3. **Tenant Admins (REQUIRED):**
   - At least one `tenant_admin` user per tenant
   - Each tenant admin MUST belong to their tenant
4. **Regular Users (REQUIRED):**
   - At least one regular `user` per tenant
5. **Projects & Tasks (REQUIRED):**
   - At least one project per tenant
   - At least one task per project

CRITICAL: ALL seed data credentials MUST be documented in `submission.json` under the `testCredentials` section. The evaluation script will use EXACT credentials from your submission.json file.

Verification: After `docker-compose up -d`, the evaluation script will:

1. Wait for services to be healthy (max 60 seconds)
2. Verify database connection via health check endpoint
3. Query database to verify seed data exists
4. Test login with seed credentials (superadmin@system.com / Admin@123)

Health Check Requirements:

partnr

- Migrations have completed
- Seed data has been loaded (if applicable)

- Health check MUST fail if database is not ready
- This allows evaluation script to know when application is ready for testing

---

STEP 6: DOCUMENTATION & DEMO

Step 6.1: Code Documentation

Task 6.1.1: README.md

File Location: `README.md` in project root

Required Sections:

1. **Project Title and Description**
   - Clear project name
   - 2-3 sentence description
   - Target audience
2. **Features List**
   - Bullet points of key features
   - Minimum 8 features listed
3. **Technology Stack**
   - Frontend technologies with versions
   - Backend technologies with versions
   - Database
   - Docker and containerization tools
4. **Architecture Overview**
   - System architecture description
   - Include architecture diagram image
5. **Installation & Setup**
   - Prerequisites

- Step-by-step local setup instructions

partnr

- How to start backend
- How to start frontend

6. **Environment Variables**
   - List all required environment variables
   - Explain purpose of each
7. **API Documentation**
   - Link to API documentation
   - OR brief list of main endpoints

Task 6.1.2: API Documentation

File Location: `docs/API.md` OR use Swagger/Postman

Options:

1. **Markdown Documentation**
   - List all 19 APIs
   - For each: Method, Endpoint, Auth required, Request body, Response format
2. **Swagger/OpenAPI**
   - Generate Swagger docs from code
   - Host at `/api/docs` endpoint
3. **Postman Collection**
   - Export Postman collection JSON
   - Include in repository
   - Document how to import

Minimum Required:

- All 19 APIs documented
- Request/Response examples for each
- Authentication explained

Step 6.2: Video Demo

Task 6.2.1: Demo Video

partnr

Required Content:

1. **Introduction**
    - Your name
    - Project title
    - Brief overview (30 seconds)
2. **Architecture Walkthrough**
    - Show architecture diagram
    - Explain multi-tenancy approach
    - Explain data isolation
    - Technology stack overview (1-2 minutes)
3. **Running Application Demo**
    - Show application running via `docker-compose up -d`
    - **Tenant Registration:**
        - Register new tenant
        - Show subdomain creation
        - Login as tenant admin
    - **User Management:**
        - Add new user to tenant
        - Show users list
        - Demonstrate role-based access
    - **Project & Task Management:**
        - Create new project
        - Add tasks to project
        - Assign tasks to users
        - Update task status
        - Show dashboard with statistics
    - **Multi-Tenancy Demonstration:**
        - Login to different tenant
        - Show data isolation
        - Attempt unauthorized access (show 403 error)

4. **Code Walkthrough**

partnr

- Highlight key files:
  - Database schema
  - Authentication middleware
  - Tenant isolation logic
  - One API endpoint code (2-3 minutes)

Technical Requirements:

- Clear audio (use microphone)
- Screen recording with narration
- Show both frontend and backend code
- No editing required (single take is fine)
- Face cam optional

Submission:

- YouTube link in README.md
- YouTube link submitted separately in submission form (not in submission.json)

SUBMISSION REQUIREMENTS

Submission Package:

1. GitHub Repository (Mandatory)

- Repository must be public
- All code committed with meaningful commit messages
- Minimum 30 commits showing development progress
- Branch structure (optional): main, develop

2. Submission JSON File

File Name: `submission.json` in repository root

MANDATORY: This file contains ONLY test credentials for automated evaluation. Documentation, demo video, and tech stack are collected separately in the submission form.

Purpose: The evaluation script uses this file to obtain test credentials for testing all API endpoints and functionality.

Format:

partnr

```
      email": "superadmin@system.com",
      "password": "Admin@123",
      "role": "super_admin",
      "tenantId": null
    },
    "tenants": [
      {
        "name": "Demo Company",
        "subdomain": "demo",
        "status": "active",
        "subscriptionPlan": "pro",
        "admin": {
          "email": "admin@demo.com",
          "password": "Demo@123",
          "role": "tenant_admin"
        },
        "users": [
          {
            "email": "user1@demo.com",
            "password": "User@123",
            "role": "user"
          },
          {
            "email": "user2@demo.com",
            "password": "User@123",
            "role": "user"
          }
        ],
        "projects": [
          {
            "name": "Project Alpha",
            "description": "First demo project"
          },
          {
            "name": "Project Beta",
            "description": "Second demo project"
          }
        ]
      }
    ]
  }
}
```

Field Descriptions:

testCredentials (MANDATORY):

- **superAdmin:** Credentials for super admin user (MUST have `tenantId: null`)
- **tenants:** Array of tenant objects, each containing:
  - `name` : Tenant organization name

- `subdomain` : Tenant subdomain (used for login)

partnr

- `admin` : Tenant admin user credentials
- `users` : Array of regular user credentials for this tenant
- `projects` : Array of project names/descriptions (optional, but recommended for testing)

Important Notes:

- **ALL seed data credentials MUST be documented** - evaluation script will use EXACT credentials from this file
- You can choose your own credentials, but MUST document them all
- Include at minimum: one super admin, one tenant with admin, at least one user per tenant
- Include projects if you want evaluation script to test project/task endpoints
- Evaluation script will test login with these credentials and use the resulting JWT tokens to test all API endpoints

---

Common Mistakes

1. Data Isolation Failures

- **Mistake:** Not filtering by `tenant_id` in API queries, allowing users to access other tenants' data.
- **Solution:** Always filter queries by `tenant_id` from JWT token. Never trust client-provided `tenant_id` in request body.
- **Example:** When listing projects, use `WHERE tenant_id = ?` with the tenant_id from the authenticated user's JWT token.

2. Super Admin Implementation

- **Mistake:** Creating super_admin with a `tenant_id` , or requiring super_admin to belong to a tenant.
- **Solution:** Super admin users have `tenant_id = NULL` . They can access all tenants but don't belong to any tenant.
- **Example:** In the users table, super_admin record: `tenant_id = NULL, role = 'super_admin'` .

3. Tenant ID Source Confusion

- **Mistake:** Getting `tenant_id` from JWT token when creating tasks (should come from project).
- **Solution:** For tasks, get `tenant_id` from the associated project, not from JWT. This ensures tasks always match their project's tenant.

- **Example:** `SELECT tenant_id FROM projects WHERE id = ?` before creating a task.

partnr

- **Mistake:** Making email globally unique instead of unique per tenant.
- **Solution:** Use composite unique constraint: `UNIQUE(tenant_id, email)`. Same email can exist in different tenants.
- **Example:** `user@example.com` can exist in tenant A and tenant B, but not twice in tenant A.

5. Missing Transaction Handling

- **Mistake:** Creating tenant and admin user separately, causing partial failures.
- **Solution:** Wrap tenant registration in a database transaction. Rollback if either tenant or user creation fails.
- **Example:** Use `BEGIN TRANSACTION`, create both records, then `COMMIT`. On error, `ROLLBACK`.

6. Authorization Logic Errors

- **Mistake:** Allowing tenant_admin to update subscription plans or tenant status.
- **Solution:** Check user role before allowing field updates. Tenant admins can only update `name`. Super admins can update all fields.
- **Example:** If `role !== 'super_admin'` and request includes `subscriptionPlan`, return 403.

7. Cascade Delete Issues

- **Mistake:** Not handling cascade deletes properly, causing foreign key constraint errors.
- **Solution:** Either use CASCADE delete in database schema, or manually set foreign keys to NULL before deletion.
- **Example:** When deleting a user, set `assigned_to = NULL` in all tasks before deleting the user.

8. Subscription Limit Checks

- **Mistake:** Not checking limits before creating users or projects.
- **Solution:** Query current count, compare with `max_users` or `max_projects` from tenant record, return 403 if limit reached.
- **Example:** `SELECT COUNT(*) FROM users WHERE tenant_id = ?` then compare with `tenant.max_users`.

9. JWT Token Payload

- **Mistake:** Not including `tenantId` in JWT payload, or including sensitive data.
- **Solution:** Include only `{userId, tenantId, role}` in JWT. Never include passwords or other sensitive data.
- **Example:** `jwt.sign({ userId: user.id, tenantId: user.tenant_id, role: user.role }, secret)`.

10. Session Table Usage

partnr

sessions, delete on logout.

- **Example:** For JWT-only, logout just returns success. Client removes token from storage.

11. Audit Logging Gaps

- **Mistake:** Not logging important actions like user creation, deletion, project updates.
- **Solution:** Log all CREATE, UPDATE, DELETE operations in `audit_logs` table with user_id, tenant_id, action type, and entity details.
- **Example:** After creating a user, insert into audit_logs: `{action: 'CREATE_USER', entity_type: 'user', entity_id: newUserId}`.

12. API Response Format Inconsistency

- **Mistake:** Returning different response structures for different endpoints.
- **Solution:** Always use consistent format: `{success: boolean, message?: string, data?: object}`.
- **Example:** Success: `{success: true, data: {...}}`. Error: `{success: false, message: "Error description"}`.

13. Frontend Authentication State

- **Mistake:** Not checking token validity on app load, or not handling token expiry.
- **Solution:** On app load, call `GET /api/auth/me` to verify token. If 401, redirect to login. Implement auto-logout on token expiry.
- **Example:** In React, use `useEffect` to check auth on mount, store auth state in context.

14. Missing Input Validation

- **Mistake:** Not validating request body fields on backend, trusting frontend validation only.
- **Solution:** Always validate on backend. Check required fields, data types, email format, password strength, enum values.
- **Example:** Validate `subdomain` is alphanumeric, `email` matches email regex, `password` is min 8 chars.

15. CORS Configuration

- **Mistake:** Not configuring CORS properly, causing frontend API calls to fail. Using `localhost` in Docker network instead of service name.
- **Solution:** Configure CORS to allow requests from frontend URL. Use environment variable for frontend URL. In Docker, use service name (`http://frontend:3000`), not `localhost`.
- **Example:** `app.use(cors({ origin: process.env.FRONTEND_URL }))` where `FRONTEND_URL=http://frontend:3000` in Docker.

FAQs

partnr

A: Super admin users have `tenant_id = NULL` in the users table. They are not associated with any tenant. When a super_admin makes API calls, their JWT token will have `tenantId: null` and `role: 'super_admin'`. Your authorization logic should check: if `role === 'super_admin'`, allow access to any tenant's data. Otherwise, enforce tenant isolation.

Q2: Should I use the sessions table or JWT-only authentication?

A: The sessions table is optional. You can implement JWT-only authentication, which is simpler and stateless. If you use JWT-only:

- Login: Generate JWT and return it
- Logout: Just return success (client removes token)
- Token validation: Verify JWT signature and expiry on each request

If you use sessions table:

- Login: Create session record and return token
- Logout: Delete session record
- Token validation: Check if session exists and hasn't expired

Q3: When creating a task, should I get tenant_id from JWT or from the project?

A: Always get `tenant_id` from the project, not from JWT. This ensures data integrity:

1. Verify the project exists and belongs to the user's tenant
2. Get `tenant_id` from that project: `SELECT tenant_id FROM projects WHERE id = ?`
3. Use that `tenant_id` when creating the task

This prevents scenarios where a user might try to create a task with a different tenant_id.

Q4: Can the same email exist in multiple tenants?

A: Yes. Email uniqueness is per-tenant, not global. The constraint should be `UNIQUE(tenant_id, email)`. This means:

- `user@example.com` in tenant A is allowed
- `user@example.com` in tenant B is also allowed
- But `user@example.com` cannot exist twice in tenant A

Q5: What happens when a tenant reaches their user limit?

A: When a tenant_admin tries to add a new user:

1. Count current users: `SELECT COUNT(*) FROM users WHERE tenant_id = ?`

2. Get tenant's max_users: `SELECT max users FROM tenants WHERE id = ?`

partnr

Same logic applies for projects with `max_projects`.

Q6: How do I handle tenant_id in queries for super_admin?

A: Super admin should be able to query across all tenants. In your query logic:

```
if (user.role === 'super_admin') {
  // No tenant_id filter - get all records
  query = "SELECT * FROM projects";
} else {
  // Filter by tenant_id
  query = "SELECT * FROM projects WHERE tenant_id = ?";
}
```

For super_admin accessing a specific tenant's data (like `GET /api/tenants/:tenantId`), verify the tenant exists but don't check if super_admin belongs to it.

Q7: What should be included in the JWT token payload?

A: Include only essential, non-sensitive data:

- `userId`: User's unique identifier
- `tenantId`: User's tenant ID (null for super_admin)
- `role`: User's role ('super_admin', 'tenant_admin', or 'user')

Do NOT include:

- Password or password hash
- Email (can be fetched from database if needed)
- Any other sensitive information

Q8: How should I handle the tenant registration transaction?

A: Tenant registration must be atomic:

1. Begin database transaction
2. Create tenant record
3. Create admin user record (with tenant_id pointing to new tenant)
4. If both succeed: Commit transaction
5. If either fails: Rollback transaction

This ensures you never have a tenant without an admin, or an admin user without a tenant.

Q9: What's the difference between tenant_admin and super_admin permissions?

partnr

- **tenant_admin:** Can manage users, projects, and tasks within their own tenant only. Can update tenant name but not subscription plan or status.
- **super_admin:** Can access and manage all tenants. Can update any tenant's subscription plan, status, and limits. Can list all tenants. Not associated with any specific tenant.

Q10: Should I validate subdomain format?

A: Yes. Subdomain should:

- Be alphanumeric (letters and numbers only)
- May include hyphens but not at start/end
- Be lowercase
- Have reasonable length (e.g., 3-63 characters)
- Be unique across all tenants

Example validation: `/^[a-z0-9]([a-z0-9-]*[a-z0-9])?$/` and length check.

Q11: How do I handle the "sessions" table if I'm using JWT-only?

A: You can either:

1. Skip creating the sessions table entirely
2. Create it but leave it empty (for future use)
3. Create it and use it optionally (hybrid approach)

The specification says it's optional. If you use JWT-only, you don't need to insert records into sessions table. Just implement JWT validation in your authentication middleware.

Q12: What should I log in the audit_logs table?

A: Log all important actions:

- User creation, update, deletion
- Project creation, update, deletion
- Task creation, update, deletion
- Tenant updates (especially subscription plan changes)
- Login/logout events (optional but recommended)

Include: `tenant_id`, `user_id` (who performed action), `action` (e.g., 'CREATE_USER'), `entity_type` (e.g., 'user'), `entity_id` (the ID of affected entity), `ip_address` (if available).

Q13: How should I handle password hashing?

A: Use a secure hashing algorithm:

partnr

- Never store plain text passwords
- Never use MD5 or SHA1 (they're insecure)
- When verifying passwords, use the library's compare function (don't hash and compare strings)

Example: `bcrypt.hash(password, 10)` for hashing, `bcrypt.compare(password, hash)` for verification.

Q14: What happens when a user is deleted?

A: Handle cascade deletes based on your database schema:

- If foreign keys have CASCADE delete: Related records are automatically deleted
- If not: Manually handle relationships:
  - Tasks: Set `assigned_to = NULL` for tasks assigned to deleted user
  - Projects: If user created projects, decide: delete projects or transfer ownership
  - Audit logs: Keep them (they're historical records)

The specification says "Cascade delete related data OR set assigned_to to NULL in tasks" - choose one approach consistently.

Q15: How do I implement pagination for the "List All Tenants" endpoint?

A: Use offset-based or cursor-based pagination:

```
SELECT * FROM tenants
WHERE (status = ? OR ? IS NULL)
  AND (subscription_plan = ? OR ? IS NULL)
ORDER BY created_at DESC
LIMIT ? OFFSET ?;
```

Calculate:

- `totalPages = Math.ceil(totalCount / limit)`
- `currentPage = page`
- `offset = (page - 1) * limit`

Return pagination metadata in response.

Q16: Should I validate tenant status before allowing operations?

A: Yes. Check tenant status in critical operations:

- Login: Reject if tenant status is 'suspended'
- Creating resources: Reject if tenant status is 'suspended' or 'trial' (if trial has restrictions)

- Active tenants: Allow all operations

partnr

Q17: How do I handle the "Edit Project" functionality mentioned in frontend?

A: The specification mentions "Edit" buttons but doesn't define an "Update Project" API. You have two options:

1. Create `PUT /api/projects/:projectId` endpoint (recommended)
2. Use the existing delete and create flow

If creating the endpoint, follow the pattern:

- Authorization: tenant_admin OR project creator
- Request body: `{name?, description?, status?}`
- Verify project belongs to user's tenant
- Update and return updated project

Q18: What should the health check endpoint return?

A: The health check should verify:

- API server is running
- Database connection is active
- Return status of both

Example response:

```
{
  "status": "ok",
  "database": "connected",
  "timestamp": "2024-06-15T10:30:00Z"
}
```

If database is down, return:

```
{
  "status": "error",
  "database": "disconnected",
  "timestamp": "2024-06-15T10:30:00Z"
}
```

Q19: How should I structure the frontend API calls?

A: Create a centralized API service:

- Base URL from environment variable

partnr

- Interceptor to handle 401 errors (redirect to login)
- Separate functions for each API endpoint

Example:

```
// api.js
const api = axios.create({ baseURL: process.env.REACT_APP_API_URL });
api.interceptors.request.use(config => {
  config.headers.Authorization = `Bearer ${getToken()}`;
  return config;
});
export const login = (email, password, subdomain) => api.post('/auth/login', {...});
```

Q20: What's the expected behavior when a tenant_admin tries to delete themselves?

A: Return 403 Forbidden with message "Cannot delete yourself". The specification explicitly states "tenant_admin cannot delete themselves". Check if `userId === currentUserId` before deletion, and reject if true.
