Software Engineer Screening Task

Overview

This is a technical assessment designed to evaluate your skills with modern full-stack development. You'll build a simplified project management system that demonstrates proficiency with our core tech stack and architectural patterns.

Task: Mini Project Management System

Build a multi-tenant project management tool with the following features:

Requirements

Backend (Django + GraphQL) -

1. Core Data Models

1. Create Django models for:
2. `Organization` (name, slug, contact_email)
3. `Project` (organization-dependent with name, status, description, due_date)
4. `Task` (project-dependent with title, description, status, assignee_email)
5. `TaskComment` (linking to tasks with content, author_email, timestamp)

2. API Layer

1. GraphQL schema with queries and mutations for:
    a. Listing projects for an organization
    b. Creating/updating projects and tasks
    c. Adding comments to tasks
    d. Basic project statistics (task counts, completion rates)

3. Multi-tenancy

1. Implement organization-based data isolation
2. Ensure proper data separation
3. Add organization context to all operations

Frontend (React + TypeScript) -

1. Project Dashboard

1. List view of projects with status indicators
2. Create/edit project form with validation
3. Responsive design using TailwindCSS

2. Task Management

1. Task board or list view
2. Add/edit tasks with status updates
3. Comment system for tasks

3. GraphQL Integration

1. Apollo Client setup with error handling
2. Optimistic updates for mutations
3. Proper cache management

4. UI Components

1. Modern, clean component design
2. Proper TypeScript interfaces
3. Loading states and error handling
4. Basic animations/transitions

Technical Stack

1. **Backend**: Django 4.x, Django REST Framework, GraphQL (Graphene), PostgreSQL
2. **Frontend**: React 18+, TypeScript, Apollo Client, TailwindCSS
3. **Database**: PostgreSQL (Docker/local setup)

Deliverables

1. **GitHub Repository** with clean commit history
2. **Setup Instructions** (README with installation steps)
3. **API Documentation** (endpoint list + GraphQL schema)
4. **Demo** (screenshots or brief video)
5. **Technical Summary** (decisions made, trade-offs, future improvements)

Sample API Structure

```
# Example models - expand as needed
class Organization(models.Model):
        name = models.CharField(max_length=100)
        slug = models.SlugField(unique=True)
        contact_email = models.EmailField()
        created_at = models.DateTimeField(auto_now_add=True)
```

```
class Project(models.Model):
        organization = models.ForeignKey(Organization, on_delete=models.CASCADE)
        name = models.CharField(max_length=200)
        description = models.TextField(blank=True)
        status = models.CharField(max_length=20, choices=STATUS_CHOICES)
        due_date = models.DateField(null=True, blank=True)
        created_at = models.DateTimeField(auto_now_add=True)

class Task(models.Model):
        project = models.ForeignKey(Project, on_delete=models.CASCADE)
        title = models.CharField(max_length=200)
        description = models.TextField(blank=True)
        status = models.CharField(max_length=20, choices=TASK_STATUS_CHOICES)
        assignee_email = models.EmailField(blank=True)
        due_date = models.DateTimeField(null=True, blank=True)
        created_at = models.DateTimeField(auto_now_add=True)
```

Frontend Component Examples

```
// Example interfaces
interface Project {
        id: string;
        name: string;
        description: string;
        status: 'ACTIVE' | 'COMPLETED' | 'ON_HOLD';
        taskCount: number;
        completedTasks: number;
        dueDate?: string;
}

interface Task {
        id: string;
        title: string;
        description: string;
        status: 'TODO' | 'IN_PROGRESS' | 'DONE';
        assigneeEmail: string;
        dueDate?: string;
}
```

Evaluation Criteria

Must Have (70%)

1. ✅ Working Django models with proper relationships
2. ✅ Functional GraphQL API with organization isolation

3. ✅ React components with TypeScript
4. ✅ Apollo Client integration
5. ✅ Clean code structure and organization

Should Have (20%)

1. ✅ Form validation and error handling
2. ✅ Basic test coverage
3. ✅ Responsive UI design
4. ✅ Proper database migrations
5. ✅ Mock external integrations

Nice to Have (10%)

1. ✅ Advanced GraphQL features (subscriptions, complex filtering)
2. ✅ Comprehensive testing
3. ✅ Docker containerization
4. ✅ Performance optimizations
5. ✅ Advanced UI features (drag-and-drop, real-time updates)

Key Focus Areas

1. **Architecture**: Clean separation of concerns, proper abstractions
2. **Data Modeling**: Efficient relationships, proper constraints
3. **API Design**: RESTful principles, GraphQL best practices
4. **Frontend Patterns**: Component composition, state management
5. **Documentation**: Clear setup instructions, API documentation

Bonus Points

1. Real-time features using WebSockets/subscriptions
2. Advanced filtering and search capabilities
3. Mobile-responsive design
4. Accessibility considerations
5. Performance monitoring/logging
6. CI/CD setup

Submission Instructions

Submit within **48 hours**:

1. **GitHub repository** (public or invite access)
2. **Readme** file with instructions to set up and start the application locally

Questions?

Feel free to ask about:

1. Specific feature requirements
2. Technical implementation approaches
3. Scope clarifications
4. Setup or tooling questions

This task evaluates your ability to build production-ready applications with modern tools while demonstrating good engineering practices. Focus on quality over quantity - a well-executed subset is better than a rushed complete implementation.

Good luck! 🚀