**Real-Time Collaborative Task Management System**

This documentation will guide you through setting up and running the backend of your real-time collaborative task management system for remote teams using Django. The frontend is handled by embedding JavaScript directly into the HTML pages, so no frontend installation is required.

**Table of Contents**

1. **Project Overview**
2. **Setup Instructions**
   * Prerequisites
   * Installation Guide
   * Database Configuration
   * Running the Application
3. **System Design**
   * Database Schema
   * API Structure
   * Real-time Communication Mechanism
   * Scalability Approach
4. **Backend Implementation**
   * Core Services
   * RESTful APIs
   * Real-time Updates
   * Authentication & Authorization
5. **Frontend Implementation**
   * JavaScript Integration
   * Real-time Updates on Frontend
   * Task Management UI
6. **Additional Features**
   * Analytics Dashboard
   * Plugin System

**7. Design Choices & Trade-offs**

**1. Project Overview**

This project is a real-time collaborative task management system designed for remote teams. It allows team members to create, assign, and track tasks in real-time, enhancing productivity and collaboration. The backend is built using Django, and the frontend leverages plain JavaScript within HTML pages.

**2. Setup Instructions**

**Prerequisites**

Ensure you have the following installed**:**

* Python 3.12
* Django 5.1
* SQLite (or any preferred relational database**)**

**Installation Guide**

1. Create a Virtual Environment

python -m venv env

source env/bin/activate

1. Install Required Packages
2. Run Migrations

python manage.py migrate

1. Create a Superuser

python manage.py createsuperuser

1. Run the Development Server

python manage.py runserver

**3. System Design**

**Database Schema**

The database schema is designed to handle tasks, users, and real-time updates efficiently. The core tables include:

* **Users:** Stores user credentials and profile data.
* **Tasks:** Stores task details, including title, description, assignee, status, and timestamps.
* **Projects:** (Optional) Stores project data if tasks are grouped under projects.

**API Structure**

The API is structured as follows:

* **Authentication:** Endpoints for user login, logout, and token management.
* **Tasks:** CRUD operations for managing tasks.
* **Projects:** (Optional) CRUD operations for managing projects.
* **Real-time Updates:** Endpoints for fetching real-time data.

**Real-time Communication Mechanism**

For real-time communication, Django Channels can be used to implement WebSockets. This allows for real-time task updates and notifications. However, for simplicity, the current setup uses AJAX polling to fetch updates.

**Scalability Approach**

To ensure scalability, the following approaches are considered:

* **Database Indexing:** Key fields such as task\_id and user\_id are indexed to improve query performance.
* **Caching:** Use Django's caching framework to store frequently accessed data.
* **Load Balancing:** Use a load balancer to distribute requests across multiple servers in production.

**4. Backend Implementation**

**Core Services**

The core backend services are implemented using Django views and models. The services include:

* **Task Management:** Handles task creation, updating, deletion, and retrieval.
* **User Management:** Manages user authentication and profile data.

**RESTful APIs**

The APIs are built using Django REST Framework (DRF) and are organized as follows:

* **Tasks API:** Supports CRUD operations.
* **Projects API:** (Optional) Supports CRUD operations for projects.
* **Authentication API:** Manages user sessions and tokens.

**Real-time Updates**

To achieve real-time updates, WebSockets can be implemented using Django Channels. The initial implementation, however, uses AJAX polling to keep the UI in sync with the backend.

**Authentication & Authorization**

Authentication is handled using Django's built-in authentication system with token-based authorization via DRF. Each API request must include a valid token in the headers.

**5. Frontend Implementation**

**JavaScript Integration**

The frontend uses plain JavaScript embedded in HTML pages. The JavaScript handles DOM manipulation, AJAX requests, and real-time updates.

**Real-time Updates on Frontend**

AJAX polling is used to periodically fetch the latest data from the server and update the UI.

**Task Management UI**

The UI is designed to be intuitive and responsive, allowing users to create, assign, and manage tasks with ease. Drag-and-drop functionality can be implemented using libraries like Sortable.js.

**6. Additional Features**

**Analytics Dashboard**

A basic analytics dashboard can be implemented using Django and JavaScript. The dashboard will display key metrics such as task completion rates and team productivity.

**Plugin System**

A simple plugin system can be implemented to allow integration with external tools such as time tracking or file attachments. This system will include an API for registering and managing plugins.

**10. Design Choices & Trade-offs**

**Django for Backend**

**Choice:** Django was chosen for its robustness, security features, and rapid development capabilities.

**Trade-offs:** While Django's comprehensive features may introduce a learning curve, it ensures a solid foundation for scalability and maintainability.

**JavaScript for Frontend**

**Choice:** Plain JavaScript was chosen to keep the frontend lightweight and avoid dependencies on modern frameworks.

**Trade-offs:** This approach may limit the ability to create complex UIs easily, but it reduces the overhead and keeps the system simple