Project Design Decisions & Architectural Overview

1. Introduction

This document outlines the key design decisions, architectural patterns, and technological choices made during the development of this project. The goal is to provide a clear rationale for the structure of the application, ensuring maintainability, scalability, and a clear separation of concerns.

The system is a Django-based web application designed to send notifications to users smartly based on their preference(email, sms, in-app). It includes features for user accounts, threaded conversations, and asynchronous user notifications via email and SMS.

2. Core Philosophy & Guiding Principles

The architecture was guided by the following principles:

- **Modularity and Separation of Concerns:** The project is divided into distinct Django apps, each with a single, well-defined responsibility. This makes the codebase easier to understand, maintain, and test.
- **Scalability:** The architecture is designed to handle growth. The use of asynchronous task queues and a robust database like PostgreSQL are key decisions to ensure the application remains performant as user load increases.
- **Developer Experience & Automation:** Tedious and error-prone manual setup steps are automated wherever possible. This includes the initial setup of user roles and permissions.
- **Asynchronous Operations:** User-facing operations should be fast. Any process that might take time (like sending an email or SMS) is offloaded to a background worker to avoid blocking the main request-response cycle and degrading the user experience.

3. Architectural Components

3.1. Django App Structure

The project is organized into three primary Django apps: account, thread, and notification.

- **account App:**
  - **Responsibility:** Manages all aspects of user authentication, user profiles, registration, and authorization (permissions and groups).
  - **Rationale:** Centralizing user management into a single app is a standard

    Django practice. It decouples user logic from the application's other features, allowing for changes to the authentication system (e.g., adding social auth) without impacting the thread or notification apps.

- **thread App:**
    - **Responsibility:** Contains the core business logic of the application, including models for threads, and comments. It handles the creation, retrieval, updating, and deletion (CRUD) of thread and comment.
    - **Rationale:** This app represents the main feature set of the project. Isolating it allows the core domain logic to evolve independently. For example, adding features like thread tagging or reactions would be contained entirely within this app.
- **notification App:**
    - **Responsibility:** Manages the sending of notifications to users. It defines the logic for *when* a notification should be sent and *what* it should contain.
    - **Rationale:** Notification logic can become complex. By placing it in its own app, we decouple it from the actions that trigger it. The thread app can simply call a function in the notification app (e.g., send_new_post_notification()) without needing to know *how* the notification is sent (email, SMS, push notification, etc.). This makes the notification system highly pluggable and easy to extend.

3.2. Asynchronous Task Handling: Celery

- **Technology:** Celery with a PostgreSQL broker.
- **Decision:** We chose Celery for handling asynchronous tasks, primarily for sending email and SMS notifications.
- **Rationale:**
    1. **Non-Blocking UX:** Network requests to external email (SMTP) and SMS gateways can be slow and unreliable. Executing these tasks synchronously would force the user to wait, leading to a poor experience. Celery offloads these tasks to a separate worker process, allowing the web server to respond to the user immediately.
    2. **Reliability & Retries:** Celery has built-in mechanisms to retry failed tasks. If an SMS gateway is temporarily down, Celery can be configured to attempt sending the message again later, increasing the overall reliability of the notification system.
    3. **Broker Choice:** We chose **RabbitMQ** as our message broker. It is a mature, robust, and feature-complete message broker specifically designed for high-performance enterprise messaging systems.

This choice provides:

- **Decoupling:** It completely separates the messaging infrastructure from our primary database (PostgreSQL). This allows each component to be scaled, managed, and optimized independently.
- **High Throughput & Low Latency:** RabbitMQ is engineered to handle a large volume of messages with minimal delay, which is ideal for a growing application.
- **Advanced Features:** It offers flexible routing options, message queueing, and delivery acknowledgements, making it a powerful and reliable choice for production environments.

3.3. Notification System Mocking

- **Technology:** Django's console.EmailBackend and a custom SMS "console" backend.
- **Decision:** During development and testing, real emails and SMS messages are not sent. Instead, their content is printed to the console (celery console).

3.4. Authorization: "Content Creator" Group & Signals

- **Technology:** Django's built-in auth system (Groups and Permissions) and Django Signals.
- **Decision:** A user group named "Content Creator" is created programmatically, and the necessary model permissions (e.g., add_thread, add_comment) are assigned to it. This process is automated using signals.
- **Rationale:**
  1. **Role-Based Access Control (RBAC):** Managing permissions on a per-user basis is not scalable. By creating a "Content Creator" role (Group), we can assign a set of permissions to the role once. Then, we can grant users the ability to create content simply by adding them to this group. This is far more maintainable.
  2. **Automation with Signals:** We use the post_save signal to trigger the creation of the group and its permissions as well as user default notification preferences. This is a robust way to handle initial data setup.

3.5. Database

- **Technology:** PostgreSQL
- **Decision:** PostgreSQL was chosen as the primary database for the application.
- **Rationale:**

1. **Robustness & Reliability:** PostgreSQL is a production-grade, open-source object-relational database known for its reliability, feature robustness, and data integrity.
2. **Advanced Features:** It supports a wide range of advanced data types and features that may become useful as the application grows (e.g., full-text search, JSONB fields).
3. **Ecosystem:** It has excellent support within the Django and Python communities, and as mentioned above, its ability to serve as a Celery broker simplifies our technology stack.