Nova 210 SE: Backend Documentation

This document describes the backend modules and generic ideas of the Nova 210 SE project.

General structure

Backend uses django and daphne to provide RESTful API for the frontend.

In the project root exists django's default entrance point, manage.py. This file is used to run django commands and start the server in debug env.

In production build, daphne is used to serve the backend, as it is faster and more reliable than django's default server.

We use Sqlite3 as the database, Django ORM is used to interact with the database; We use InMemoryChannelLayer for the websocket layer.

Module structure and explanation

The directory structure of the backend project is as follows:

- `backend/` : Root directory of the backend project
  - `manage.py` : Django's default entrance point for running commands and starting the server in debug environment
  - `Dockerfile` : Dockerfile for building the backend image
  - `requirements.txt` : Python requirements file, states the top-level dependencies of the project
  - `start.sh` : Docker image entrypoint script
  - `backend/` : Django project directory
    - `settings.py` : Django project settings file
    - `asgi.py` : Django ASGI configuration file
  - `data/` : Directory for storing sqlite3 database
  - `main/` : Main application directory, details written below

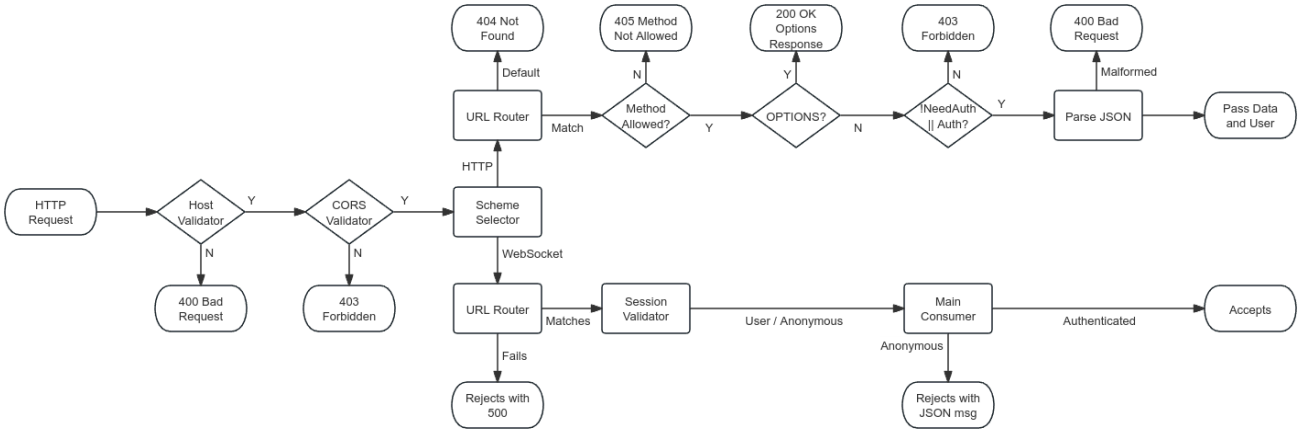
Main project files are shown below:

- `migrations/` : Directory for storing database migrations
- `models/` : Directory for storing Django models
  - `__init__.py` : Includes all models, acts as an entry point to make all models as a whole.
  - `_user.py` : User model, stores user information

    - `_friend_group.py` : FriendGroup model, stores friend group information
    - `_friend.py` : Friend model, stores friend information
    - `_friend_invitation.py` : FriendInvitation model, stores friend invitation information
    - `_chat.py` : Chat model, stores private / group chat information
    - `_user_chat_relation.py` : UserChatRelation model, stores user-specific chat information
    - `_chat_message.py` : ChatMessage model, stores chat message information
    - `_chat_invitation.py` : ChatInvitation model, stores chat invitation information
- `views/` : Directory for storing Django views
    - `__init__.py` : Empty file to make the directory a package
    - `user.py` : User views, handles user-related operations, including login / register / get info / etc.
    - `friend_group.py` : FriendGroup views, handles friend group-related operations, including create / list / delete / etc.
    - `friend.py` : Friend views, handles friend-related operations, including invite / list / delete / etc.
    - `chat.py` : Chat views, handles chat-related operations, including create / list / delete / set owner / etc.
    - `generate_avatar.py` : Utility to generate random avatars
    - `api_utils.py` : Foundamental API utilities, for example API decorators
- `ws/` : Directory for the main websocket consumer
    - `__init__.py` : Defines the consumer class itself
    - `_dispatcher.py` : Receive channel notifications and handle side-effects
    - `_notification_channel.py` : Define channels and which channel a session should join
    - `urls.py` : Defines the websocket URL routing
    - `action.py` : Defines client-to-server actions
    - `notification.py` : Defines server-to-client notifications
- `tests/` : Directory for storing unit tests and integration tests. Contents are self-explanatory
- `__init__.py` : Empty file to make the directory a package
- `exceptions.py` : Custom exceptions for the backend
- `urls.py` : Main URL routing file, includes all URL routes of the backend
- `apps.py` : Django app configuration file

Understanding authentication mechanism

The backend uses HTTP Session to authenticate users. When any request arrives, authenication process looks like:



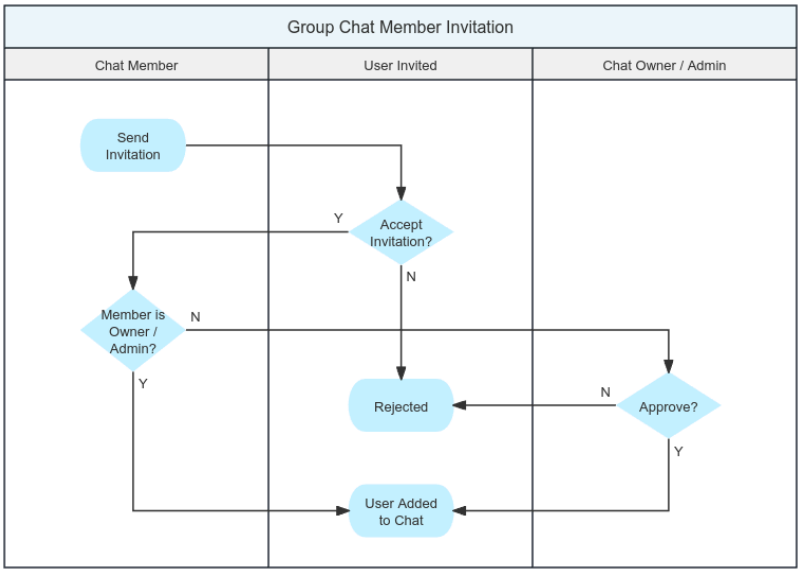
A similiar process is used for websocket messages to ensure the message is valid JSON.

Understanding complex operations

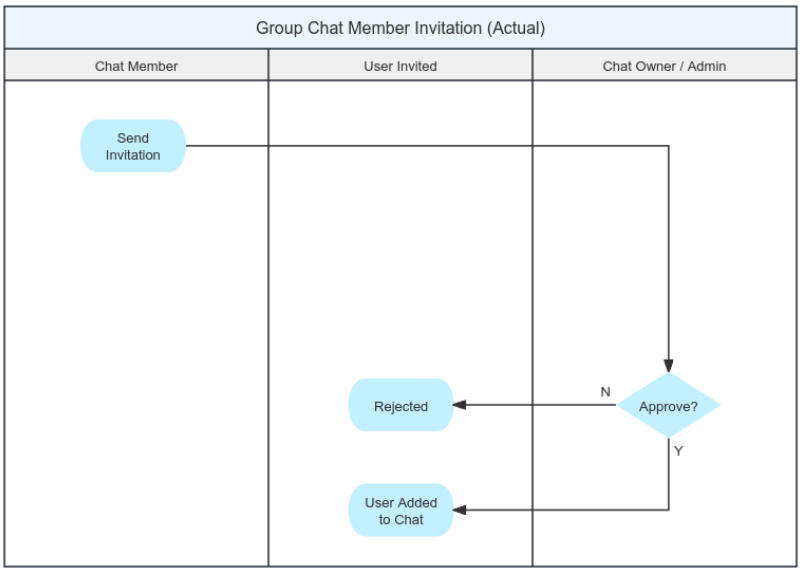
Most operations in this software are straightforward, but some operations are complex and require multiple steps to complete. They are elaborated here to better understand the logic and prevent maintenance issues.

Group chat member invitation

The ideal chat member invitation process is as follows:



For simplicity, we have implemented a simplified version of the process:



We would like to implement the ideal process in the future, if possible.

User deletion

Quite a lot of operations are required to maintain a clean database before a user can be deleted. The process is as follows:

- All friends currently online should be notified that the user is to be deleted, so that the friendship and the corresponding no longer exists.
- All private chat with the user in it should be deleted (as standard friend deletion process is not called here), to avoid private chat with only one user.
- All owned group chat will be deleted (by CASCADE strategy), so all member in these chats should be notified of the deletion of chat.
- All owned group chat is actively deleted so that "user quit chat" notification will not be sent to those chats.
- All group chat that the user is still in should be notified that the user is leaving the chat.
- All friendship is deleted actively (although as per CASCADE strategy, this is not necessary).
- All active sessions of the user is notified to log out.
- The AuthUser is deleted.
- The User is deleted.

From here on, the function returns, but the following things will happen:

- User will be removed from chat member list and chat admin list
- All owned FriendGroup is deleted (by CASCADE strategy)
- All sent message transfered sender to #DELETED user
- User will be removed from Message read list
- All pending friend invitation and chat invitaton is deleted (by CASCADE strategy)
- All related UserChatRelation is deleted (by CASCADE strategy)

Now the user is completely deleted from the database without a chance to recover.

Where to find more information

Detailed comments and documentation is written in the source code. Please refer to the source code for more information.

For updated and exact struct backend will return in an API, check each model's `to_{}_struct` method.