

# CS6650 Final Project - WeeChat

Dec 2021

Tianyu Tan, Yung-Shin Chien, Alex Yang, Yeqing Huang

## Github Links

Frontend: [https://github.com/AZYDEVE/weechat\\_frontend](https://github.com/AZYDEVE/weechat_frontend)

Backend: <https://github.com/neu-6650/final>

## Video Links

<https://www.youtube.com/watch?v=ROlqEhg6qZE>

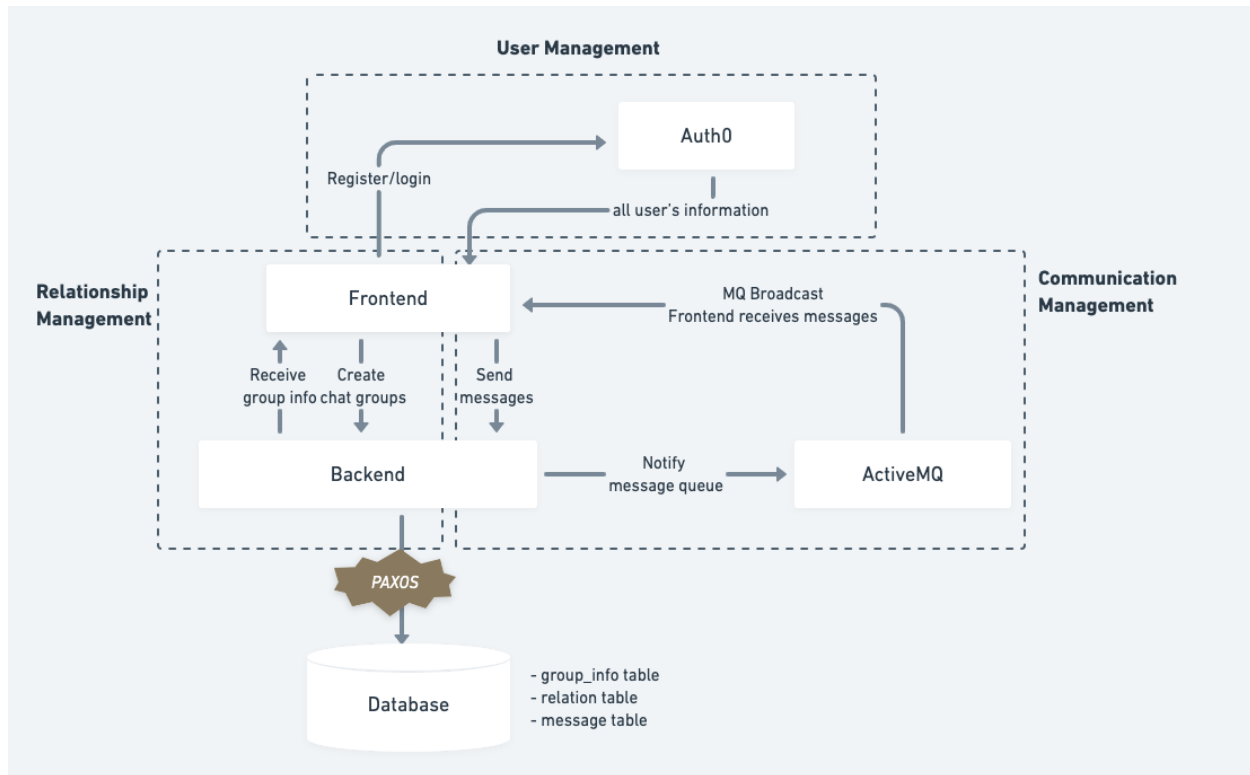
## Introduction

In this project, we would like to build a chat web application that allow friends to chat with each other, like WeChat or WhatsApp. We applied the concepts we've learned from the course to make it robust. This application adopts the publish-subscribe pattern to achieve asynchronous communication and uses PAXOS to make the service fault-tolerant in a distributed system.

## Architecture Overview

The high-level design of this project is shown in the diagram below. As planned in the project proposal, this application has three modules: user management, relationship management, and communication management.

User management is the part that we do not implement a distributed design pattern by ourselves. We use a third-party service called Auth0 to provide the basis registration and authentication features. When a user has logged in to the website, he could choose to create new chat groups or just subscribe to existing groups to receive messages. Because we place the implementation details of PAXOS in the backend, the frontend user will not feel a difference when the server that he originally connects to fails. The application will try to find another functional server for this client automatically.



<https://whimsical.com/workflow-PxayGA1HRKAEgM1kPdazSp>

## Tech stacks

**Frontend:** React, Node.js Express

**Backend:** Springboot

**Database:** MySQL

**Third-party tools:** ActiveMQ, Auth0, Docker, React WebSocket, Swagger, Lombok, Fastjson, MyBatis-Plus, Hutool

## Features

### User Registration/Login

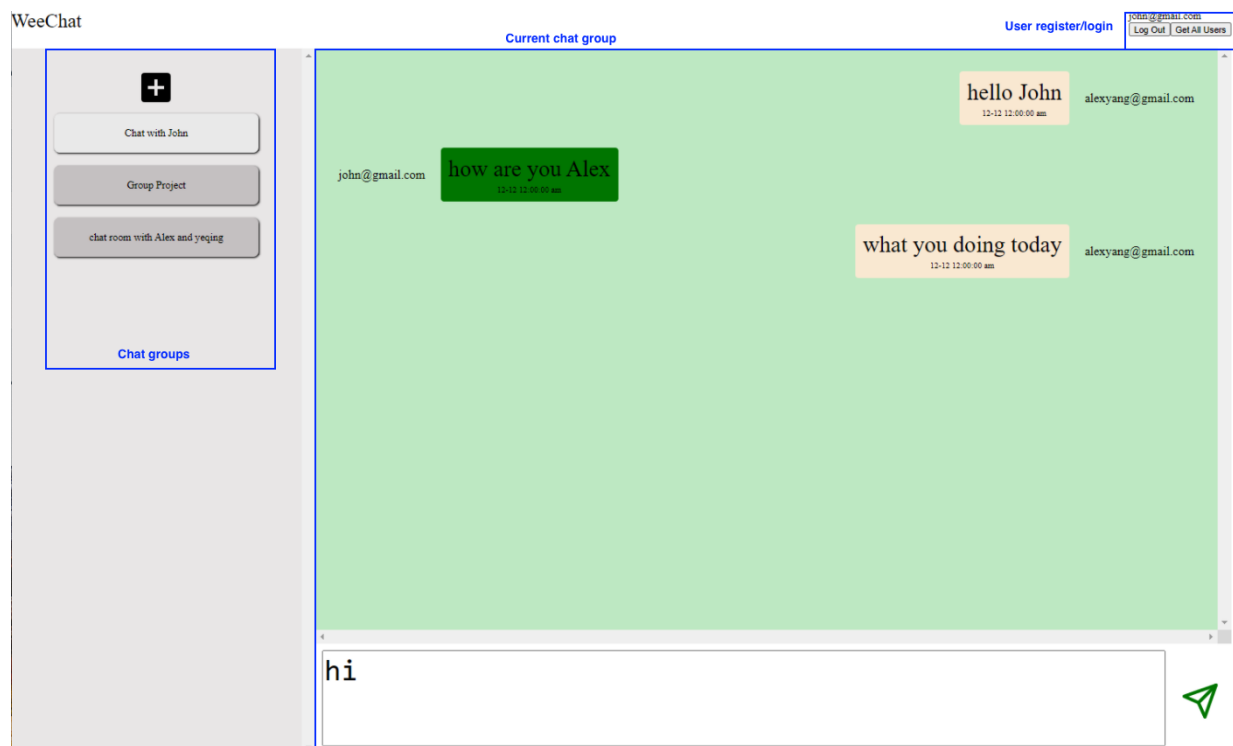
A user needs to log in to chat with others. We want to minimize this part because it is not the focus of this project. We used a third-party service called *Auth0* for user register/login/logout. Users can click on the login button on the chat page. It will redirect the users to auth0's URL for registration/authentication. All the user information and tokens are handled by auth0. We call auth0's API to get the registered users' information.

## Group Chat

We use JMS *publish and subscribe model* for the asynchronous group communication. We choose not to distinguish a 1-to-1 chat from a group chat. In other words, a 1-to-1 chat is a group chat with only two users. A group is a topic in the context of the publish/subscribe messaging.

To create a new group or chat room, the user can click on the add button on the chatroom navigation bar. Then the user needs to type in a group name and select one or more registered users from the dropdown menu. The chosen users will automatically join this group, but they won't receive messages until they subscribe to this group. For the creators, this newly created group will be shown on the navigation bar automatically. However, the new chat room would only show on other corresponding members' chat room listings when they refresh the page.

Users can view the message history of a specific chat group and receive instant messages from other members in the chat by clicking the chat room label in the navigation bar. The click of the label sends a get request to the server to receive the message history of the chat room. It also subscribes to the message provider to listen to new messages associated with the chat room ID. When the user selects a different chat room, the application will subscribe to a new ID and unsubscribe the previous chat room ID from the message provider.



In the current active chat group, the user can view the messages along with the sender name and the message timestamp in the active chat group. All the chat-related features are handled by the WebSocket from the frontend and the message queue.

## Relationship Management

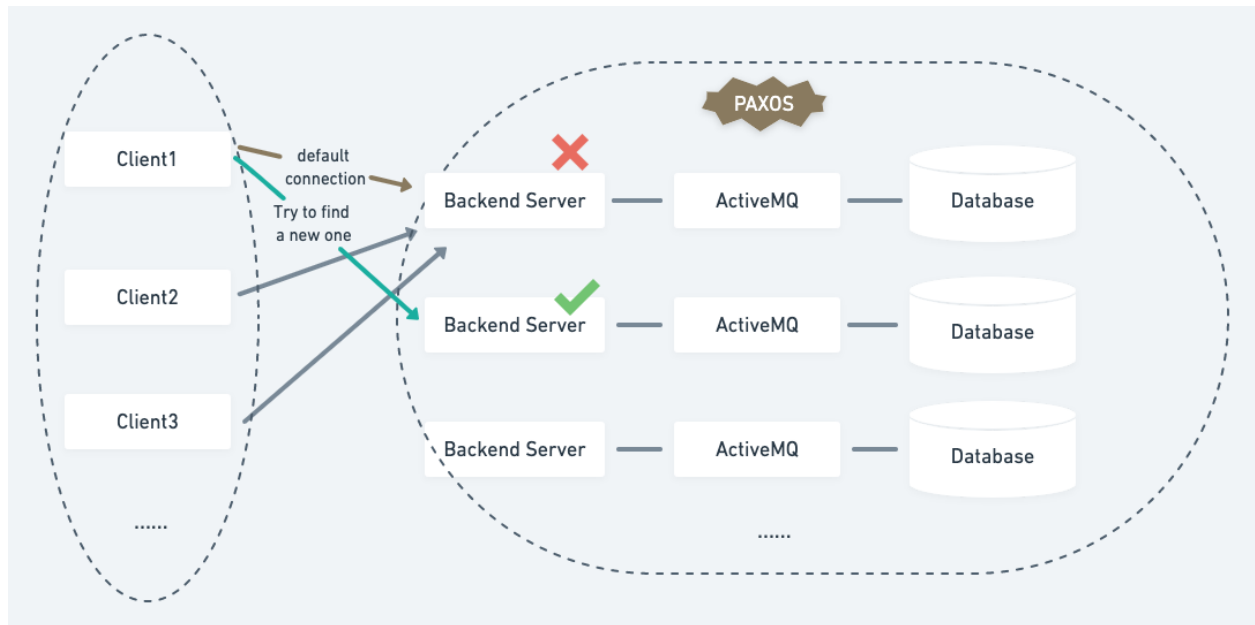
Based on the chat feature, we can tell that we need to record all the created chat groups information in the database. The SQL database holds 3 tables:

1. **group\_info**: used for creating and querying groups
2. **relation**: the relationship table between groups and users
3. **message**: it contains the sender\_id, group\_id, text message, and a timestamp

## Algorithms

The key algorithms we used in this project is publish-subscribe messaging pattern and PAXOS. From this project, we learned how to utilize ActiveMQ to implement more complex ideas. When using PAXOS, we extended the design choices based on Project 4 and incorporated the concepts of distributed consensus, group communication and fault-tolerance.

When we design this project, we want to make this chat application robust as required in the project, i.e. It should be able to continue operation even if one of the participant nodes crashes. The diagram below shows the distributed nodes in our system. Each backend server is related to a message queue and its own database. When the frontend client tries to send a message, it will connect to one server by default. If this server cannot function well and the frontend receives a status code that is not 200, it knows that the backend server fails. Then we use a for loop to try to find another available server for it. As long as there is some backend server running, this client can connect to it. If all servers fail (which is not likely to happen), we can debug it and the error logs will be shown in the browser's console.



<https://whimsical.com/distributed-3FFDDPcCzibguRgnkZMzmo>

#### *How PAXOS is implemented in the backend:*

When a proposer receives a request from a client, it sends requests to all acceptors bound to it and checks whether there is any unfinished proposal that takes a majority. If it is, the proposer finishes that proposal first and then comes back to its original proposal. Otherwise, it attaches the original operation and asks the acceptors to accept it. If an acceptor finds that the new proposal is not the same one which last visited it, it will check the value. After it accepts, this proposal is finished so it can clear the record. Otherwise, keep the record of this operation and leave it to a later proposal. When a proposal is approved, it should be broadcast to all learners so that the result can be saved.

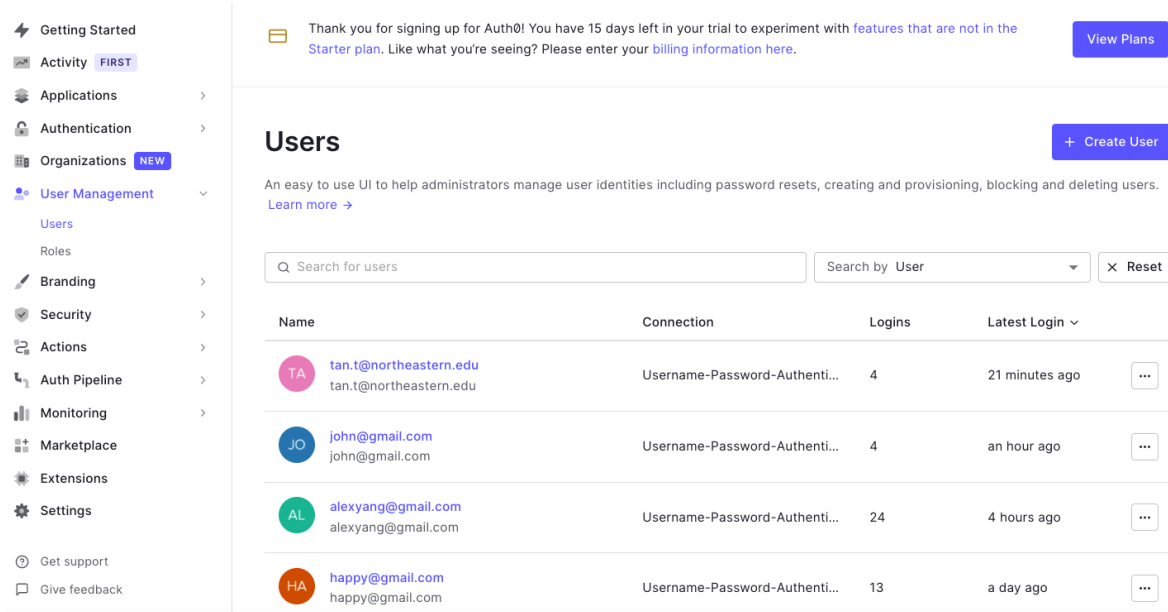
- For the fail simulation, we used a thread way. When accept or promise is called, it has 20 percent of chance to fall into a sleep and result in a timeout.
- The bind methods are in proposer class. When a server is instantiated, its proposer ask the designated acceptors and learners to bind to it, then its acceptor and learner bind to the designated proposers. In that way, when a new server is instantiated, we can customize which servers it should connect with.
- Value is encapsulated into an object called 'Operation' which get us rid of parsing the string value.
- We assume that an acceptor is always a learner so that acceptor only works in the vote phase and the executing work is only done by learners. Otherwise, when a learner is also an acceptor, the operation may be executed twice.

## Tools









Below screenshots show how we utilize third-party libraries/tools to develop this project.

### Auth0

Auth0 provides a dashboard to configure our application and shows all the users information with an administrative access.

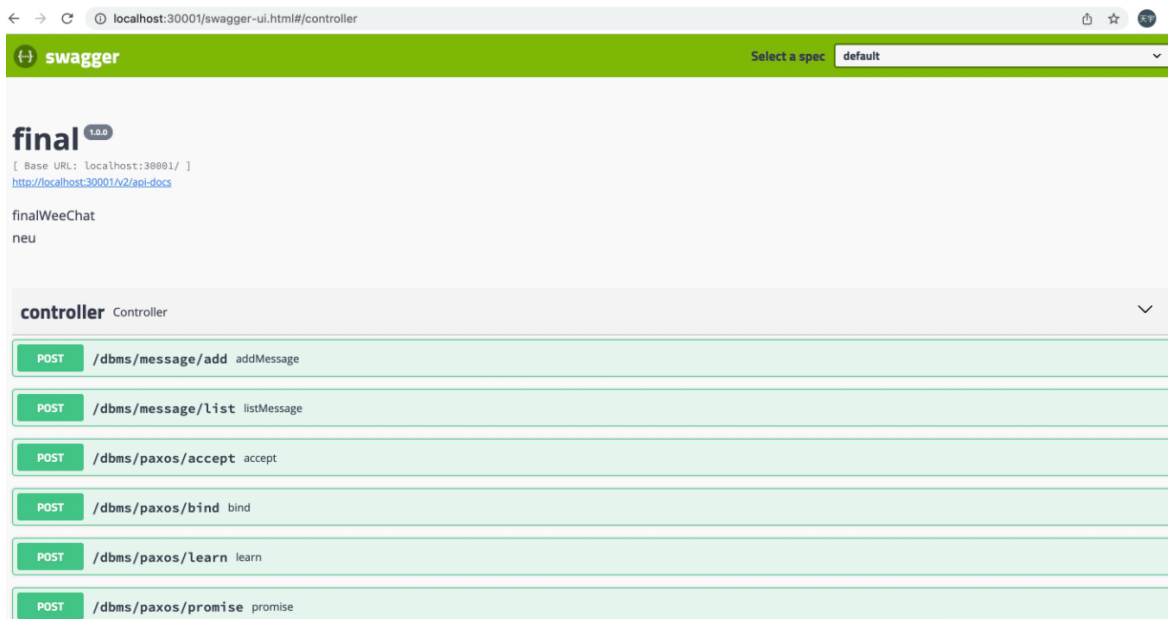


The screenshot shows the Auth0 Users dashboard. On the left is a sidebar with navigation links: Getting Started, Activity (FIRST), Applications, Authentication, Organizations (NEW), User Management (selected), Roles, Branding, Security, Actions, Auth Pipeline, Monitoring, Marketplace, Extensions, Settings, Get support, and Give feedback. The main content area has a header with a thank-you message and a 'View Plans' button. Below this is a 'Users' section with a '+ Create User' button and a description: 'An easy to use UI to help administrators manage user identities including password resets, creating and provisioning, blocking and deleting users. Learn more →'. A search bar is present with the text 'Search for users' and a dropdown menu set to 'Search by User', with a 'Reset' button. Below the search bar is a table of users.

Name	Connection	Logins	Latest Login	
 <a href="#">tan.t@northeastern.edu</a> tan.t@northeastern.edu	Username-Password-Authenti...	4	21 minutes ago	
 <a href="#">john@gmail.com</a> john@gmail.com	Username-Password-Authenti...	4	an hour ago	
 <a href="#">alexxyang@gmail.com</a> alexxyang@gmail.com	Username-Password-Authenti...	24	4 hours ago	
 <a href="#">happy@gmail.com</a> happy@gmail.com	Username-Password-Authenti...	13	a day ago	

### Swagger UI

In the backend, we use swagger-ui to display and manage all the backend APIs.



The screenshot shows the Swagger UI interface in a web browser. The address bar shows 'localhost:30001/swagger-ui.html#/controller'. The interface has a green header with the 'swagger' logo and a 'Select a spec' dropdown menu set to 'default'. Below the header, it says 'final 1.0.0' and provides the base URL 'http://localhost:30001/v2/api-docs'. The main content area is titled 'finalWeeChat neu' and shows a list of API endpoints under the 'controller' section. Each endpoint is displayed in a green box with a 'POST' method and a description.

Method	Endpoint	Description
POST	/dbms/message/add	addMessage
POST	/dbms/message/list	listMessage
POST	/dbms/paxos/accept	accept
POST	/dbms/paxos/bind	bind
POST	/dbms/paxos/learn	learn
POST	/dbms/paxos/promise	promise

## ActiveMQ

We use ActiveMQ to observe all the topics and the number of connected consumers.

Name ↕	Number Of Consumers	Messages Enqueued	Messages Dequeued	Operations
ActiveMQ.Advisory.MasterBroker	0	1	0	Send To Active Subscribers Active Producers Delete
ActiveMQ.Advisory.Consumer.Topic.27	0	2	0	Send To Active Subscribers Active Producers Delete
ActiveMQ.Advisory.Consumer.Topic.29	0	3	0	Send To Active Subscribers Active Producers Delete
ActiveMQ.Advisory.Producer.Topic.29	0	2	0	Send To Active Subscribers Active Producers Delete
ActiveMQ.Advisory.Connection	0	6	0	Send To Active Subscribers Active Producers Delete
ActiveMQ.Advisory.Consumer.Topic.28	0	5	0	Send To Active Subscribers Active Producers Delete
ActiveMQ.Advisory.Topic	0	4	0	Send To Active Subscribers Active Producers Delete
24	0	0	0	Send To Active Subscribers Active Producers Delete
27	0	0	0	Send To Active Subscribers Active Producers Delete
28	1	0	0	Send To Active Subscribers Active Producers Delete
29	1	1	2	Send To Active Subscribers Active Producers Delete

## Future Improvements

One thing that we can improve in this project is a specific server failure case. When a backend server fails, the frontend will only notice it when the client tries to send a message, otherwise the frontend webpage remains the same. The frontend won't know if the backend functions well and there is just no new messages or the backend server has already failed. One potential solution is that we could let the frontend send a ping message periodically to the backend in order to detect this failure earlier.