Chats

Kabeer Jaffri <kabeer@kabeers.network>

Architecture white paper

This paper describes the problems and solutions we came up with for this project

Table of contents:

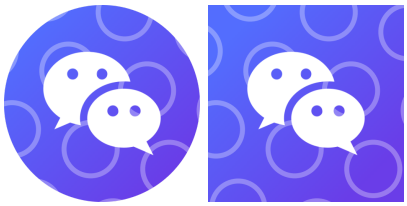
Introduction

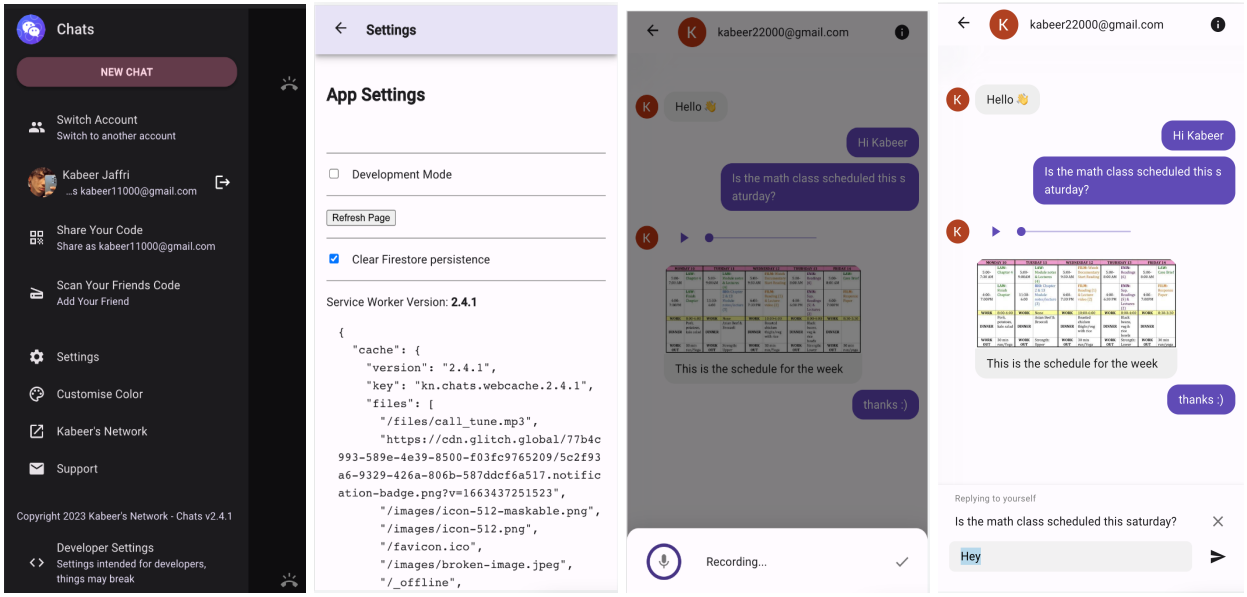
About the Chat service

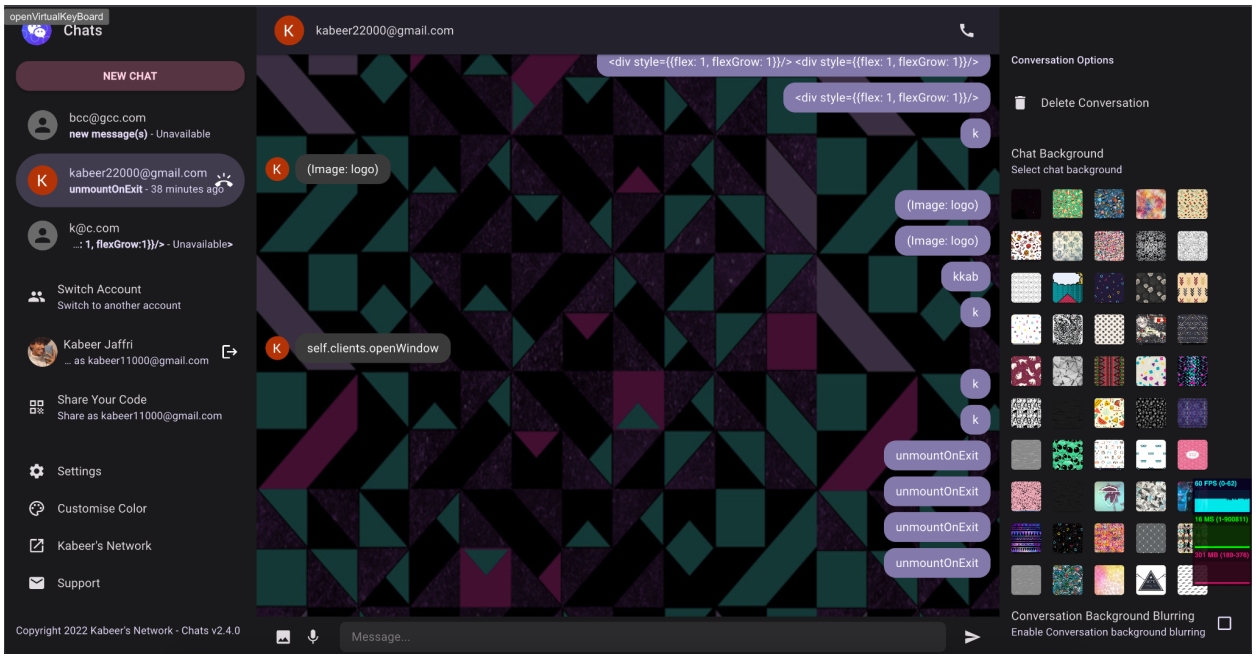
Creating a fully-fledged, production-grade, open-source email-messenger app called "Chats." This service allows you to send text messages to anyone via email, even if the other person hasn't signed up for the service. To get started, I forked a GitHub project called "chakra-chat" for the front-end of this service. It was a barebones NextJS app that used Firebase, providing a basic chat messenger experience. However, this was not sufficient for the goals of the service, so I incrementally added functionality and new features to it. To keep costs low for the backend, I adopted a microservice architecture, as described in the backend portion of this paper.

Gallery and showcase

UI and other misc images from the Chats service







User Interface and front-end service

Introduction

As you have read in the introduction, the original fork was barebones and unfinished, incrementally adding features.
The first thing I implemented was an email notification service with Sendgrid. This enables us to send alerts to users who haven't even signed up for Chats (one-way communication).
Next, I did a complete UI rewrite, transforming the simple app written with chakra UI to have a Material 3-compliant UI. For this, I used the popular React MUI library but unfortunately has no support for Material 3, Google's latest addition to the Material design specification.
I programmed logic and custom themes for color-picking and dynamic palette generation, as well as wrote custom CSS for individual UI elements.

Integrating Firebase and Firestore

I wanted to make the application sign-up as frictionless as possible, firebase provided easy-to-implement authentication services, and I used this feature inside the NextJS front-end.
The other core functionality required for any working messenger app is a real-time database. Fortunately, Firebase had me covered over there too. Doing all the heavy lifting such as scaling and managing a real-time database Firestore. Their free tier was generous and powerful.

I implemented it inside the NextJS app with the help of their JavaScript library and another external library called react-firebase-hooks. This made working with the Firestore inside React straightforward. However, as the application progressed I ran into performance and rendering issues as I made use of the react context API to implement everything.

In a later update, I migrated everything to use only the Firebase library and completely get rid of react-firebase-hooks to improve rendering performance.
This was a hassle considering I had to rewrite the entire context API-based architecture into a stateless system managed by zustand.

Theme Scheme and Material 3

The theming system used in this application is based on React MUI's powerful underlying theming API. I created a theme scheme generator, and a context provider to manage it, this way the app theme became dynamic. So much so that It can even change conversation to conversation for an ambient and personalized experience for the user

Performance tooling and testing

Going from an app prototype in NextJS to a production-grade service was a long way. The top pain points I faced were in performance and reliability. For A/B testing and app error reporting I used my custom-written tools, for measuring and accurately reporting crashes. The other thing we calculated was script-load times and core web vitals for a specific subset of our user base. Another pain point was the unreliability of using Firebase, especially in a complicated application like this. We ran into a lot of performance issues and rendering problems for example, firebase wouldn't play along easily when we combined server-side rendering into the application, often running into proxy and reinitialization issues, as Firebase configuration objects could simply not be proxied and were reinitialized on the client.

Query performance and reliability with react was also a significant issue, where documents loaded were not served from the cache, and updates would cause unnecessary re-rendering inside the app degrading the performance.

Web Performance API helped a lot in calculating timings for Firestore and other complicated functions inside the application.
One other thing that we realized soon was the amount of feature segmentation even on modern browsers. Options available inside service workers sometimes changed drastically going from browser to browser, for example even with a relatively reliable system in the backend, we still recorded times when notifications were dispatched but never shown to the client, mostly due to unreliable play between the browser notification handler and the operating system. We had to resort to using polyfills when available or just detecting and feature-blocking users who didn't support them.

Nodes and services in our backend

Microservice Architecture

Our product employs a microservice architecture that includes several key components to handle its various functionalities.
These microservices were thoughtfully introduced to the Chats through regular updates. By implementing them incrementally, the development team ensured a seamless integration and transition.
The adoption of a microservice architecture, with separate components like "peer," "live," and "push," allows Chats to maintain modularity, scalability, and fault isolation. Each microservice focuses on a specific aspect of the chat app, enabling independent development, deployment, and maintenance. This decoupled approach enhances agility and flexibility, enabling Chats to adapt and evolve in response to changing user needs and technological advancements.

Kabeer Cloud Storage Services

Chats incorporate a custom-written storage engine that efficiently creates indexes of files and saves them on a Content Delivery Network (CDN). This storage engine is designed to meet the specific needs of a service of any scale, providing a reliable and fault-tolerant solution. By leveraging this custom tooling, We reduced our dependence on third-party services like Firebase, allowing for greater control and flexibility in file management.

While this service is not currently open source, its stability and fault tolerance are crucial aspects that contribute to the reliable operation of many of our products

Push manager and notification service

A NodeJS service that uses Sockets and HTTP is crucial in handling push functionality within the Chats app. This microservice is particularly robust and dependable, supporting various architectures and methods for different types of notifications throughout the service.
We first added Notification Service V1 with the Chats version 2.4.0 but with a newer release and alpha support for calls and other types of messages, service V1 was more or less retired and replaced with service V2 which supports highly dynamic message schemas and protocols, ranging from text and call alerts to update notifications and background tasks.

The push microservice's versatility allows Chats to deliver notifications efficiently, optimizing user engagement and interaction.

Remote configuration manager

Another essential component in the Chats architecture. This service acts as a robust configuration management system, similar to Firebase's Remote Config. It allows for personalized user experiences across different versions of the Chats application.

The remote configuration manager empowers the development team to dynamically adjust application settings, feature flags, and user-specific parameters without requiring app updates. This flexibility ensures that Chats can adapt to changing user preferences and market requirements swiftly. By leveraging the remote configuration manager, Chats maintains a high level of customization and tailors the user experience to meet individual needs.

Crash tracking and user analytics

We use a combination of our solutions and Google Analytics to detect how users interact with our service. Our custom crash analytics engine is written in PHP, it is responsible for detecting common and fatal errors during the runtime in production and testing phases of the chats service. We use a custom deployment toolchain to work with the Chats source code, which auto-deploys everything to their designated configurations.

Both of these services provide valuable insights into user behavior, feature adoption, and performance metrics, empowering data-driven decision-making to optimize the application for improved user engagement and satisfaction. With this integrated solution, Chats ensures a stable and reliable chat experience while proactively addressing issues and continuously enhancing the application's performance.

Peer service and RTC

The peer microservice plays a pivotal role within the Chats app's microservice architecture. Initially, in alpha version 1, it utilized PeerJS for managing the Peer-to-Peer (P2P) and RTC (Real-Time Communication) call functionality. However, as video and voice-calling features moved to their beta phase, a strategic shift was made to enhance the audio and video calling experience.

In the beta phase, the peer microservice has transitioned to using Mediasoup as a Selective Forwarding Unit (SFU) module. Mediasoup acts as an intermediary between WebRTC connections from clients, enabling reliable group calling and low-latency video and voice calling. This switch to Mediasoup has significantly improved the scalability and performance of the calling feature within Chats.

With Mediasoup, Chats can efficiently route audio and video streams between participants in group calls, ensuring synchronized communication. The SFU functionality provided by Mediasoup allows for optimized bandwidth usage, minimizing latency and maximizing the quality of audio and video calls. This shift in technology has enabled us to offer a seamless and immersive group calling experience to its users.

Native applications for Chats

Introduction

As we progressed in some aspects we outgrew functionality provided by PWA's especially feature segmentation across browsers pushed us to our decision for the creation of native apps to serve an uninterrupted experience to our users.

Our Front-end PWA did most of the functionality necessary, so we figured wrapping it inside a native app with frameworks such as Capacitor would be our best bet.

Android, IOS, and Desktop apps

We created a basic WebView-based app that provided all the core functionalities to the underlying Chats PWA, relaying a few native functions to improve performance and immersion. Reaching near native-like experiences unobstructed by browser restrictions on features such as notifications.

For Desktop we picked electron as it was the dominant and reliable framework in the market. It also supported this kind of wrapping perfectly as it already embeds Chromium

For Mobile apps, we used Ionic Framework using Capacitor for all the heavy lifting in the native application

Refrences

Firebase: [Firebase](https://firebase.google.com/) - A comprehensive platform by Google that provides a suite of services for building and managing web and mobile applications.

Next.js: [Next.js](https://nextjs.org/) - A popular React framework for building server-side rendered and statically generated web applications.

Chakra UI: [Chakra UI](https://chakra-ui.com/) - A simple and accessible component library for building user interfaces with React.

Material-UI: [Material-UI](https://material-ui.com/) - A popular React component library that implements Material Design guidelines for creating modern and visually appealing UIs.

React Firebase Hooks: [React Firebase Hooks](https://www.npmjs.com/package/react-firebase-hooks) - A set of React hooks that provide an easy way to interact with Firebase services in a React application.

React: [React](https://reactjs.org/) - A JavaScript library for building user interfaces, widely used for creating interactive and reusable UI components.

Zustand: [Zustand](https://github.com/pmndrs/zustand) - A small, fast, and scalable state management library for React applications.

Mediasoup: [Mediasoup](https://mediasoup.org/) - An open-source WebRTC media server designed to enable real-time communication in web and native applications.

Capacitor: [Capacitor](https://capacitorjs.com/) - An open-source framework that allows developers to build cross-platform mobile apps using web technologies.

Electron: [Electron](https://www.electronjs.org/) - A framework that enables building desktop applications using web technologies such as HTML, CSS, and JavaScript.

Google Analytics: [Google Analytics](https://analytics.google.com/) - A web analytics service offered by Google that tracks and reports website traffic and user behavior.
