A Project Final Review Report

on

Real-Time Voice Chat (Podcast)

Submitted in partial fulfillment of the

requirement for the award of the degree of

B. Tech in Computer Science and Engineering

GALGOTIAS UNIVERSITY

(Established under Galgotias University Uttar Pradesh Act No. 14 of 2011)

Under The Supervision of
Name of Supervisor: Ms. Vaishali Gupta
Designation: Assistant Professor

Submitted By

Anish Kumar 19SCSE1010275
Apoorav Singhal (19SCSE1180077)

SCHOOL OF COMPUTING SCIENCE AND ENGINEERING
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING /
DEPARTMENT OF COMPUTERAPPLICATION
GALGOTIAS UNIVERSITY, GREATER NOIDA
INDIA

December 2021

SCHOOL OF COMPUTING SCIENCE AND ENGINEERING
GALGOTIAS UNIVERSITY, GREATER NOIDA

CANDIDATE'S DECLARATION

I/We hereby certify that the work which is being presented in the thesis/project/dissertation, entitled **"Real-Time Voice Chat (Podcast)"** in partial fulfillment of the requirements for the award of the **B.TECH** submitted in the School of Computing Science and Engineering of Galgotias University, Greater Noida, is an original work carried out during the period of month, Year to Month and Year, under the supervision of **Ms. Vaishali Gupta, Assistant professor**, Department of Computer Science and Engineering/Computer Application and Information and Science, of School of Computing Science and Engineering , Galgotias University, Greater Noida

The matter presented in the thesis/project/dissertation has not been submitted by me/us for the award of any other degree of this or any other places.

Anish Kumar 19SCSE1010275
Apoorav Singhal (19SCSE1180077)

This is to certify that the above statement made by the candidates is correct to the best of my knowledge.

Supervisor Name : **Ms. Vaishali Gupta**
Designation: **Assistant professor**

<u>CERTIFICATE</u>

The Final Thesis/Project/ Dissertation Viva-Voce examination of **Anish Kumar 19SCSE1010275** and Apoorav Singhal 19SCSE1180077 has been held on 22nd Dec 2021 and his/her work is recommended for the award of B.TECH.

Signature of Examiner(s) **Signature of Supervisor(s)**

Signature of Project Coordinator **Signature of Dean**

Date: December 2021

Place: Greater Noida

Abstract

In recent time demand of audio book is network chat communication have seen an enormous rise in popularity over the last several years. In this article we consider MP3 audiobook applications and propose an approach to completely restyle the applications to the current mobile and multimedia scenario. This paragraph describes the process of the development of a real time voice chat application for Student. The main aim of this project is to make a platform where students can complete their literature syllabus along with doing other work such as eating, traveling, or any other work. We describe common features across these systems and highlight distinctions between them.

This article we consider MP3 audiobook applications and propose an approach to completely restyle the applications to the current mobile and multimedia scenario. Where possible, we discuss the advantages and disadvantages of different technical approaches used in these systems to support different features and functions. This project will help the authors to understand JavaScript and Web Sockets, in a better way. In this project, the author will use the most popular SPA library(React) and NoSQL databases such as MongoDB and Redis. Rest JSON APIs with Node, and many other technologies will be used in this project.

This is a web based project which help students in their studies. A study was conducted to determine the impact of the use of audiobooks with struggling readers in a school library audiobook club. The participants met weekly in the school library with the school librarian and researchers to discuss audiobooks and make reading recommendations to their peers. Standardized test data as well as pre- and post-study interviews and surveys, teacher questionnaires, parent questionnaires, and student interviews were analyzed. The findings indicated that struggling readers’ use of audiobooks had a positive impact on reading skills and attitudes toward reading.

This dissertation describes the process of the development of a chat application for developers, from a mere idea to a working cloud service.The author has built a real time platform that makes it easy to have a group con- versation between a projects’ members, share code and stay up to date with their latest repository updates.
It has given him a better understanding of AJAX and WebSocket’s, SPA libraries (React), NoSQL databases (MongoDB and Redis), REST JSON APIs with Node, Docker, and more.

Table of Contents

List of Table

List of Figures

Acronyms

B.Tech.	Bachelor of Technology
SCSE	School of Computing Science and Engineering
IT	Information Technology
CSS	Cascading Style Sheets
SASS	syntactically awesome style sheets
AJAX	Asynchronous JavaScript And XML
JSON	JavaScript Object Notation
API	Application programming interface
SRS	Software Requirement Specification
HTML	Hypertext Markup Language
UML	Unified Modeling Language

CHAPTER-1

Introduction

With the rapid development of mobile phones, mobile devices have become one of the integral part of daily activities. In recent years, chat applications have evolved and made a major change in social media because of their distinctive features that attract audiences[1]. It provides real-time messaging and offers different services including, exchange text messages, images, files and etc. Moreover, it supports cross platforms such as Android and iOS. There are currently hundred millions of users smartphone are using chat applications on monthly basis[2]. There are two types of architecture in those applications, client-server and peer-to-peer networks. In a peer-to-peer network, there is no central server and each user has his/her own data storage.

On the contrary, there are dedicated servers and clients in a client-server network and the data is stored on a central server[3]. Security and privacy in chat applications have a paramount importance but few people take it seriously. In a test done by the Electronic Frontier Foundation, most of the popular messaging applications failed to meet most security standards. These applications might be using the conversations as an information for certain purposes. Moreover, reading the private conversations is certainly unacceptable in terms of privacy.

We propose to apply a machine learning technique for the incoming task requests so as to classify the best suitable algorithm for the task request rather than randomly assigning the scheduling algorithm. Unsupervised machine learning techniques can be used here. The outcome of the proposed work leads to the selection of the best task scheduling algorithm for the input task(request).

This article we consider MP3 audiobook applications and propose an approach to completely restyle the applications to the current mobile and multimedia scenario. Where possible, we discuss the advantages and disadvantages of different technical approaches used in these systems to support different features and functions. This project will help the authors to understand JavaScript and Web Sockets, in a better way. In this project, the author will use the most popular SPA library(React) and NoSQL databases such as MongoDB and Redis. Rest JSON APIs with Node, and many other technologies will be used in this project.

This is a web based project which help students in their studies. A study was conducted to determine the impact of the use of audiobooks with struggling readers in a school library audiobook club. The participants met weekly in the school library with the school librarian and researchers to discuss audiobooks and make reading recommendations to their peers. Standardized test data as well as pre- and post-study interviews and surveys, teacher questionnaires, parent questionnaires, and student interviews were analyzed. The findings indicated that struggling readers' use of audiobooks had a positive impact on reading skills and attitudes toward reading.

Most applications only used Transport Layer Security (TLS) for securing channel, the service provider has full access to every message exchanged through their infrastructure [4]. Therefore, these messages can be accessed by attackers. Therefore to maintain protection and privacy, messages should be encrypted from sender to receiver, and no one can read messages even the service provider, in addition to protecting the local storage of the device [5]. In this paper, we focus on security, privacy and speed by proposing end-to-end security which ensures only sender and receiver can read messages without a third party. As well as storage protection and fast transfer of messages between the parties. The main contributions of this paper are the following:

1- Propose client-server mobile chat application which supports the status of the communicating parties whether online or offline.
2- Provide a friendship request service.
3- Secure key exchange, then calculate the session key.
4- Secure exchange of end-to-end messages.
5- Analysis and Test the proposed chat.

The aim of this project is to build a functional real-time messaging application for developers by using modern web technologies. Unlike most chat applications available in the market, this one will focus on devel- opers and will attempt to boost their productivity. Although we are not expecting it to have a plethora of utilities due to the limited time frame, sharing code and watching a repository will be our core features.
It will be fully open-source. Everyone will be able to dig into the code to read what is going on behind the scenes, or even contribute to the source code. So it was within our intentions to write clean, scalable code following the most popular patterns and conventions for each of the languages and relevant libraries.

1.1. Formulation of Problem

This document is organized by phases, albeit not explicitly indicated in some of the sections.
In the next section, Research and Prototypes, we will describe the process of looking up complementary information for the project in various sources. For example, which protocols to use when dealing with real time data and why. Next, we will move onto the planning and technologies, on which we do our best to have everything ready for the development of the project.

Afterwards, in the implementation section, we describe the most relevant bits of the development of the web application. We avoid commenting fragment by fragment what the code does since that can be easily checked from our GitHub repository and focus on the design decisions (why we did it one way or another, and the range of possibilities we had). The implementation section itself we do also sort the features by chronological order, rather than describing front end first and back end later. At the end of the day, it is what makes the most sense because we work in both at the same time: we wrote the server feature part first and we made sure it was working as expected by testing it with our client part working. The deployment stage comes next, where we will explain the various deploying options, we had and why we ended up using Docker for the job.

To finish, we will evaluate the most relevant dependencies that took part during the development, and also the methodology we chose, and we will list the future work tasks that were left to do, or we suspect that there is room for improvement.

1.2.1. Tool and Technology Used

In this project we are using following tools and technologies:

- **Operating System:** Mac, Window 7, Window 10, window 11
- **Machine Architecture:** 32bit and 64bit
- **Tool:** Vscode, Postman, Firefox, Live server, GitHub, Doker
- **Programming language:** JavaScript, Nodejs
- **Markup Language**: HTML
- **Style-Sheet** : CSS, SASS
- **Library:** Reactjs, web Sockets
- **Processor speed:** Minimum: 1.0 GHz, Recommended: 2.0 GHz or faster
- APIs
- socket.io

CHAPTER-2

Literature Survey

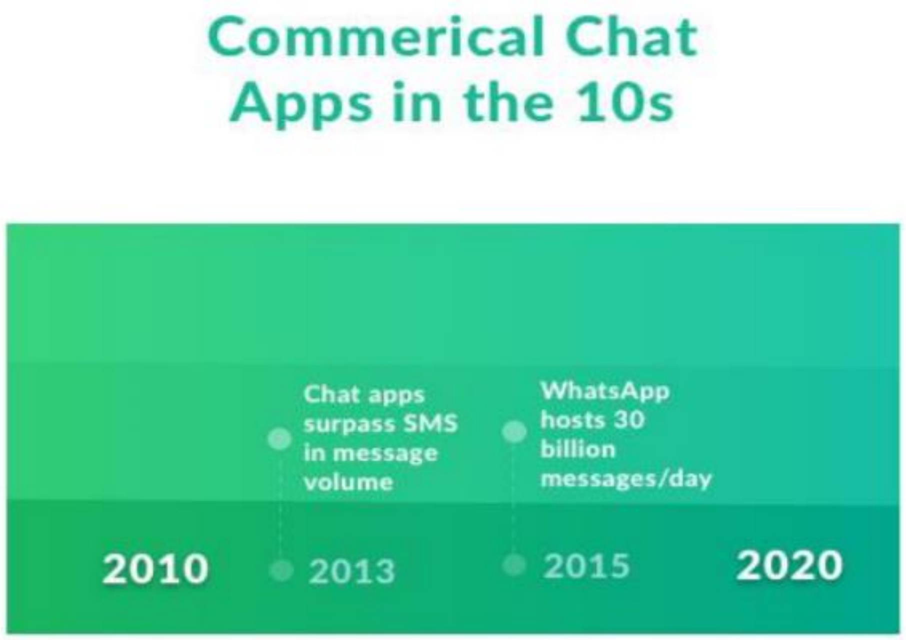

Commerical Chat Apps in the 10s
Chat apps surpass SMS in message volume
WhatsApp hosts 30 billion messages/day
2010
2013
2015
2020

Figure 1literature Servey

As we have noticed messaging apps now have more global users than traditional social networks—which mean they will play an increasingly important role in the distribution of digital information in the future. In 2016, over 2.5 billion people used at least one messaging app. That's one-third of the world's entire population, with users ranging from various age grades. Today, it's common place for offices to use a messaging app for internal communication in order to coordinate meetings, share pitch decks, and plan happy hours. And with the latest bot technology, chat apps are becoming a hub for employees to do work in their apps without leaving the chat console. With the inception of smart phones, chat apps continued to thrive; in 2013, chat apps finally surpassed SMS in message volume. By 2015, WhatsApp alone hosted 30 billion messages per day. SMS logged only 20 billion. And in the summer of 2016, Facebook Messenger hit one billion users.

Mobile Reader For Visually Disabled. The voice user interface by proposed model Tyflo recognizes specific commands. When user speaks, it maps the word from the command database, works accordingly and brings out the desired output on to the screen. For example, “Read Article” command from the user reads the article. “Move Paper Up” changes to next successive page.

Mobile gaming for Visually Challenged Children. An affordable, learning centered mobile application for visually impaired is proposed through this paper. Graphics for visually impaired is still a major headache for they will not be experiencing them completely without vibrations. This model contains three inputs: Stimuli getting, response determining and output giving. Using psychomotor activities [2] and sensors, the intellectual use of their brain is enabled. This allows them to play games with auditory control.

Audio reality mobile application that helps to navigate. This application utilizes GPS, Wi-Fi and 3G data services which enables the user to use Google map to navigate. With the help of Smart Robot (SR)[3] and sensory devices installed, when the user meets the obstacles in their path, the application senses it and shows warning. The visually challenged person also gets information about the nearby places and ways to navigate/reach to it with this application.

Display for visually impaired people in mobile phones. Background consists of color, size and location. Foreground consists of text. Auditory display is used for the foreground display. Auditory and tactile modality [4] is used for the background display. Blindfolded users performed operations very quickly when these services were offered. Foreground display becomes the dominant suggests that of representing data through auditory and tactile modalities like TTS and Braille display.

Hands free navigation system for Visually Challenged people. With the help of this voice enabled system, which is already proposed and available, one can search queries from the search engines through related keywords. It proves to be a hands-free navigation system to the visually challenged people. The voice enabled system [5] is having five different phases: parameter extraction, partitioning, sound clarification, sentence analysis and word determination, and word recognition. The major leap in developing apps for visually impaired pioneered here from this application development.

2.2. Competitive Analysis

Prior to getting started with the application development, we did some research on the current messaging platforms out there. We were looking forward to building a unique experience, rather than an exact clone of an existing chat platform. We already knew of the existence of several messaging applications, and a few chat applications that suited developers. However, never before had we done an in-depth analysis of their tools to find out whether they were good enough for developers.

Soon, we realized that none of the sites were heading in our direction. Some of them were missing features which we considered crucial, and others had opportunities for further enhancements.

Contrary to what many people think, having a few platforms around is not a necessarily a bad thing. We were able to get ideas of what to build and how and determine which technologies and strategies to use based on their experience. Often, this was as simple as checking their blogs. Companies like Slack regularly post development updates (such as performance reviews, technology comparisons, and scalability posts). Other times, we had to dig into the web to find out the different options we had and pick out the one which we considered to be the most appropriate.

- Flowdock - https://www.flowdock.com
- Gitter - https://gitter.im
- Hangouts - https://hangouts.google.com
- Matrix - http://matrix.org
- Messenger - https://messenger.com
- Rocket. Chat - https://rocket.chat
- Skype - https://web.skype.com
- Slack - https://slack.com
- Telegram - https://web.telegram.org
- WhatsApp - https://web.whatsapp.com

Gitter and Slack were the ones which focused on developers the most. Although basic things such as code sharing were also possible in some others such as Rocket. chat, it was clear that they were not targeting developers, that often resulted in a lack of tools for them. We believe that they do need a professional environment that is built for them and makes code sharing pleasant, as well as having the possibility to integrate it with their source code repository.

How will our app be different from Slack or Gitter?

At this point, it was clear that we should be narrowing down our full analysis to Slack and Gitter. The other platforms were still beneficial to extract a few general concepts, but they were far from our topic. Slack and Gitter are both popular platforms which focus on productivity and developers and have proved to do so well during a considerable amount of time. Leaving aside that our application will be open-source, which was not the case with any of them, there are a few other features that will make our application different from these two.

To start, our platform will be room-based, which is not the case with Gitter. Gitter rooms are either GitHub users or organizations (which they call communities) that in our opinion are not very flexible. Slack has a room-based system (called "teams"), but it does not have the ability to create chat rooms linked to a git or GitHub repository which is something we are trying to improve with our application. The closest it supports is bot integration (which can send messages of any kind, such as a GitHub feed), but that leads to a very cluttered chat if the repository is at least somewhat active.

Moreover, none of them have the ability to fork conversations based on a previous message, which we intend to support. At the moment, on both Gitter and Slack, a chat fork has to be done manually, which implies a lot more work for the user than having chat rooms created and destroyed at the touch of a button.

Chapter 3

Working of Project

For most web applications, communication protocols are not a subject of discussion. AJAX through HTTP is the way to go since it is reliable and widely supported. However, that is not our case. We need, albeit not in every single situation, an extremely fast communication method to send/receive messages in real time. For messaging, there are a few communication protocols available for the web. The most popular ones are AJAX, WebSocket's, and WebRTC.

AJAX is a slow approach. Not only because of the headers that have to be sent in every request, but also, and more important, because there is no way to get notified of new messages in a chat room. By using AJAX, we would have to request/pull new messages from the server every few seconds, which would result in new messages to take up to a few seconds to appear on the screen, not to say the numerous redundant requests that this would generate.

WebSocket's are a better approach. WebSocket's connections can take up to few seconds to establish, but thanks to the full-duplex communication channel, messages can be exchanged swiftly (averaging few milliseconds delay per message). Also, both client and server can get notified of new requests through the same communication channel, which means that unlike AJAX, the client does not have to send the server a petition to retrieve new messages but rather wait for the server to send them. WebRTC is the new communication protocol available for the most modern browsers (Chrome, Firefox, and Opera). It is designed for high-performance, high-quality communication of video, audio, and arbitrary data.

WebRTC does not require any server as a proxy to exchange data, other than the signaling server that is needed to share the network and media metadata (often done through WebSocket's). The fact that stream data can be exchanged between clients directly often means faster messaging and less server-side workload. WebRTC can run over TCP and UDP, but it often runs with UDP by default. Although UDP can lead to packet loss it does give a better performance which can lead to a more fluid voice or video call, and we can afford to lose a few frames when video calling.

Given the advantages and disadvantages of the three technologies, we decided to use WebSockets for real-time messaging, which guarantees us packets delivery (unlike frames on a video call, we do not want to miss out any text message), as well as having a good compatibility and being a popular and documented choice. We were going to use WebRTC for both voice and video calls[2], which ended up being part of the future work. WebRTC would make a conversation more fluid in the majority of occasions due to faster packet delivery time. When it comes to dropped voice or video packets, we do not really care about this as long as they are just a few of them.

When it comes to random requests, such as authentication, room creation or listing, AJAX is a good option. It does not require a permanent connection to the server, which results in less power usage for both the client and the server, and the requests response times are decent. However, none of these kinds of requests require an extremely rapid response. What is more, AJAX requests are so popular that the latest browsers' versions themselves already offer high-level APIs as well as the low-level legacy libraries, which makes it trivial for any programmer to fetch JSON from any remote host without using any specific library.

Project Methodology

Agile is a set of techniques to manage software development projects. It consists in:

- Being able to respond to changes and new requirements quickly.
- Teamwork, even with the client.
- Building operating software over extensive documentation.
- Individuals and their interaction over tools.

We believed it was a perfect fit for our project since we did not know most requirements beforehand. By using the Agile, we were able to focus only on the features which had the most priority at the time.

Use Cases and Scenarios

User stories are one of the primary development artifacts when working with Agile methodology. A user story is a very high-level definition of a requirement, containing just enough information so that the developers can produce a reasonable estimate of the effort to implement it[3]. Gathered from stakeholders (people, groups or organizations who are interested in the project), they show us what we have to work in. Since we were working with Agile, this list did not have to be complete before we started working on the project, but it was desirable to have at least a few items to start with so that we could establish proper feature priorities.

At the commencement of every sprint, we analyzed all user stories, estimated the value they added to the project and the amount of time they would take us doing each of them, and sorted them by descending order — placing the user stories which had the most added value and the least time cost at the top. The value was quite subjective. We gave the highest priority to features which we believed they were essential to the platform (such as instant text messages) or were very related to the chat's topic — coding. We gave them a score from 1-10.
Time cost was an estimation of how much we thought an individual story was going to take to implement. The measurement was done in days, considering each working day to be as long as 4 hours. We then translated this value as follows:

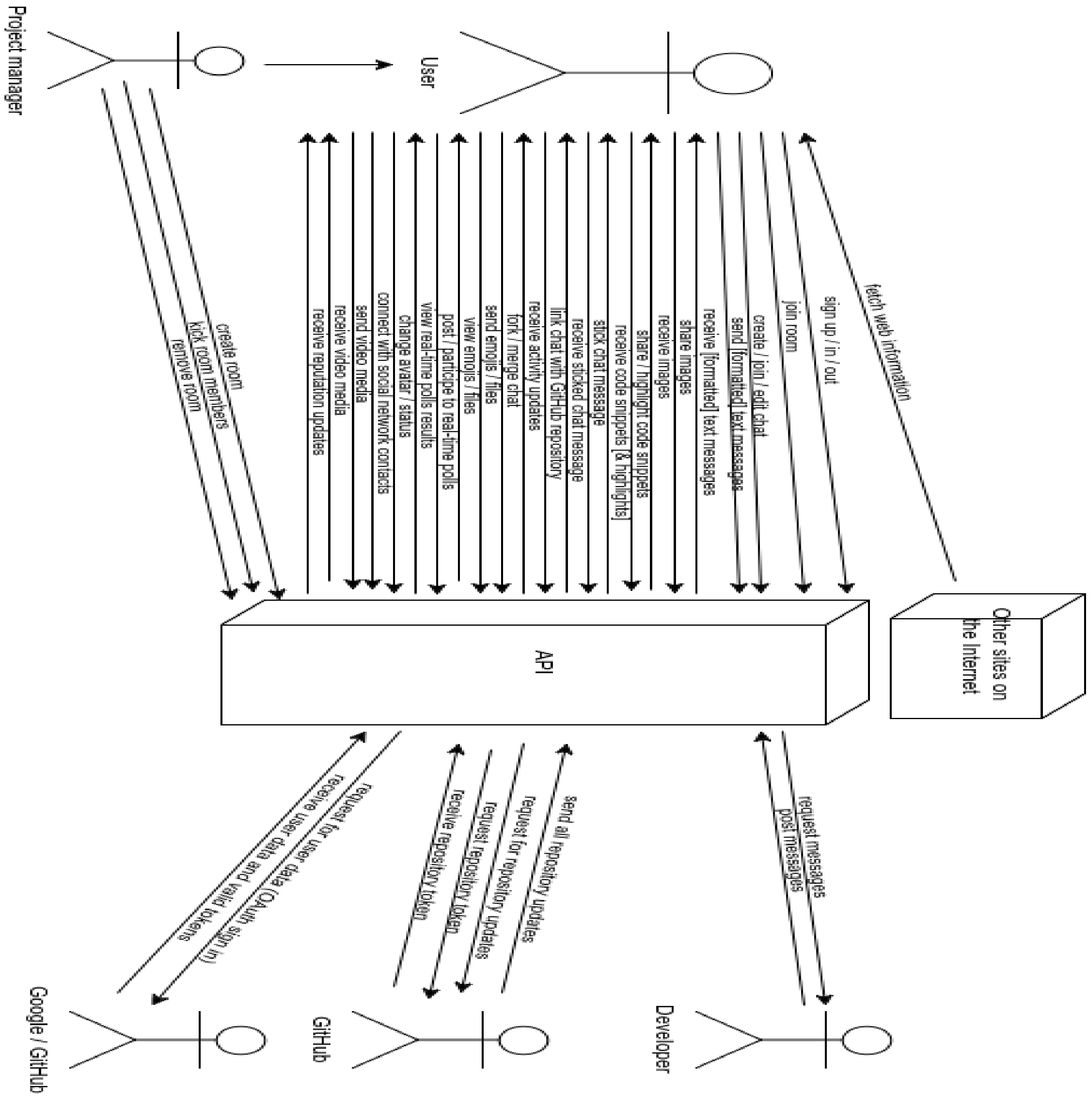

Project manager
User
create room
kick room members
remove room
receive reputation updates
receive video media
send video media
connect with social network contacts
change avatar / status
view real-time polls results
post / participe to real-time polls
view emojis / files
send emojis / files
fork / merge chat
receive activity updates
link chat with GitHub repository
receive sticked chat message
stick chat message
receive code snippets [& highlights]
share / highlight code snippets
receive images
share images
receive [formatted] text messages
send [formatted] text messages
create / join / edit chat
join room
sign up / in / out
fetch web information
API
Other sites on the Internet
request for user data (OAuth sign in)
receive user data and valid tokens
send all repository updates
request for repository updates
request repository token
receive repository token
request messages
post messages
Google / GitHub
GitHub
Developer

Figure 2use Case Diagram

Chapter 4

Results and Discussion

4.1. Technology

The architecture of the application consists of the back end and the front end, both of them having their own set dependencies (libraries and frameworks).

The front end is the presentation layer that the end user sees when they enter the site. The back end provides all the data and part of the logic, and it is running behind the scenes.

3.4.1 Back End

The "back end" refers to the logic and data layers running on the server side. In our case, the back end makes sure that the data introduced through the client application (the front end), is valid. Since the front end can be avoided or easily manipulated (the source code is available to the end user) we have to make sure that all the requests we receive are first verified by the server: the requested URI is supported, the user has the appropriate permissions, the parameters are valid, etc. If the request data is valid, we do often proceed to execute some logic accompanied by one or more database accesses.

API

Our application is all about I/O. We were looking forward a programming environ- ment which was able to handle lots of requests per second, rather than one which was proficient at handling CPU-intensive tasks. At the moment it seemed like the choice was between PHP, Python, Java, Go or Node.js. These languages have plenty of web development documentation available, and they have been widely tested by many already. The trendiest choice in 2016 was Node.js, which was exceptional for handling I/O requests through asynchronous processing in a single thread.

So we went for Node.js not only because of the performance but also because of how fast it was to implement stuff with it, contrary to other languages such as Java which are way more verbose. For web development, we would then use Express, which makes use of the powerfulness of Node.js to make web content even faster to implement.
A feasible alternative to Node.js would be Go, which is becoming popular nowadays due to somewhat faster I/O than Node.js with its Go subroutines, and unquestioned better performance when doing intensive calculations[4] (though we were not particularly looking for the last one).
Nonetheless, Go meant slower development speed. It lacked libraries as it was not as mature as Node.js and the cumbersome management of JSON made it not very ideal for our application (since the JavaScript client would use JSON all the time).

We are writing Node.js with the latest ECMAScript ES6 and ES2017 standard supported features. The development was started with ES6, but we also used a few features originally from ES2017 as soon as Node.js turned to v7.

ES6/ES2017 standards differ from the classic Vanilla JavaScript in that they have a few more language features and utilities out of the box which makes code easier to read, faster to write and reduce the need to make use of external libraries to do the most common operations. For example, Promises over callbacks or classes over functions, even though they are just syntactical sugar. A few remarkable frameworks/libraries we are using on the development of the application are:

Express: A Node.js framework which makes web development fast. It abstracts most of the complexity behind the web server and acts as an HTTP route handler. It can also render views (a sort of HTML templates with variables) but are using the front end application for this instead. By using Express, we are able to focus on the logic behind every request rather than on the request itself.

Mongoose: A MongoDB high-level library. By using objects as database models, which will later end up being the data inside our collections, it handles inserts, updates, and deletes, as well as the validation for each of its fields.

Passport: An authentication library build specifically for Node.js. By using the different login modules (one module per provider), it hides all the complexity behind OAuth, OAuth2, and OpenID. Passport commits to notifying the developer in the same way regardless of the authentication method they have chosen. We are using Passport to handle GitHub and Google authentication, as well as the local one (email + password).

Simon: An extensive testing library that has a set of useful utilities[5]: spies, stubs, and mocks. Throughout our tests, we often feel the need to know whether a certain function has been called, has been called with the right parameters, or even to fake external incoming data to ensure that we are testing solely what we want to test.

Socket.io: A JavaScript library which handles WebSocket connections. It abstracts most of the complexity behind WebSocket's, and it also provides fallback methods which work without any special configuration. Socket.io takes care of the real time updates in our application, such as sending or receiving messages.

Data storage: We believe that NoSQL is the future. Hence, we did not hesitate to choose to use NoSQL storages only. Why choosing NoSQL databases over the traditional SQL ones?

- They are more flexible: you can access nested data without having to per- form any join.
- They are faster nested data is stored in the same place and can be consulted without any additional query.
- They scale better when distributing the data over different nodes.
- There are many types of NoSQL databases which fit for different kinds of work, such as Key-Value for sessions or Document-based for complex data.

At first, we were going to go with MongoDB only, but later we realized that it would be a good idea to have Redis as well to map session keys with user identifiers.

MongoDB is a schemeless document-oriented database. It gives us the possibility to store complex data effortlessly and retrieve it straight away, without any additional query. Although the data is always stored on disk, it is very fast and highly scalable. We are using MongoDB to store any persistent data, such as user details and preferences, rooms information and chat messages. You can find more details about the MongoDB document modeling on the implementation chapter.

Redis is a key-value data structure, which uses memory storage to perform searches by a given key very quickly. Queries are performed faster than in MongoDB but there is also a higher risk of losing data, and it cannot process complex values (such as nested documents). Although Redis performs better than MongoDB, we cannot rely on it for critical data. For this reason, we are only using it to store user sessions, which in the case of loss, would only mean that the user would have to re-login to keep using our platform. Nonetheless, we expect to performance gains to be noticeable when the site is at its peak capacity because the session data is something we are are looking up in every single request to the API.

3.4.2 Front End

Having separated the server-side from the client side, a SPA (Single-Page Applica- tion) was an outstanding choice. SPAs dynamically fetch data from the API as the user is browsing the site, avoiding to refresh the whole page whenever the user has filled in a form or navigated to another part of the site.
The UX boost a SPA can get over a traditional website is very significant. It is true that it often takes longer to load for the first time, due to having to download a bigger JavaScript file chunk, but once loaded the delay between operations is minimal which leads to a more fluid User experience, and less bandwidth use in most case.

Implementing a scalable Single-Page Application by using Vanilla JavaScript only would take an enormous amount of time, since it has none of the high-level utilities that make it simple to develop one of this kind, such as a high-level HTML renderer that allows you to build elements on the fly, storage or router. Hence, it made sense to choose an actively maintained and documented framework/library to start with. At the time, the decision was between Angular, React and Vue.

Both Angular and React were being maintained by powerful corporations, Google and Facebook respectively, so we had a brief look at their documentation and developers' reviews before taking our final choice. Eventually, we chose React.

React is a very powerful library with an enormous ecosystem (you can find many utilities that were meant to be used with React). It is featured due to its fast performance and small memory consumption, which is especially useful when targeting mobile devices. Moreover, there is a plethora of documentation on its official site and around the Internet.

The library main features are:

Tree Structure:

A React page always starts with a single root component (tree node) rendered in a pre-existing HTML element on the page. Each component can have one or more children.

```
functionRootComponent(props) {
        return<h1>It works!</h1>;
}

ReactDOM.render(<RootComponent/>,document.getElementById('main'));
```

Custom DOM Elements

React does not work with HTML components directly. Instead, it uses component (which often have the same names) which will be later trans piled into HTML components.

```
<input className="foo"/>
<textarea value="123"/>
```

is transpiled to

```
<inputclass="foo"/>
<textarea>123</textarea>
```

Information is transferred down the tree through component properties.

A root component might not need to have access to external information since it is technically the one who owns the whole application.
However, children's components might need to. For example, a component which is responsible for displaying a generic input text box along with a label will need to get to know a name.

```
<form>
    <TextInput name="email"/>
    <TextInput name="address"/>
    ...
</form>
```

TextInput component

```
functiontextInput(props) {
    return(
        <fieldset>
            <labelfor="props.name">{props.name}<      /label>
            <input type="text"id="{props.name}"placeholder="{props.name}"/>
    </fieldset>
    );
}
```

Information is transferred up the tree through function references

At some point, the parent might need to get to know about information that changed on a child so that to react in one way or another. For example, in the snippet above, it might need to get to know when the value changed on the input so as to later process the form information.
Parent component

```
functioninputChanged(event) { ... }
functionParentComponent(props) { return(
<form>
<TextInput name="email"onChange={inputChanged}/>
<TextInput name="address"onChange={inputChanged}/>
...
</form>
);
}
```

TextInput component

```
functiontextInput(props) { return(
    <fieldset>
      <labelfor="props.name">{props.name}<    /label>
      <input
        type="text"
        id="{props.name}"
        placeholder="{props.name}"
        onChange={props.onChange(...)}/>
  </fieldset>
  );
}
```

React itself, contrary to Angular, is just the View in the MVC architecture. Hence, we required of additional libraries to fulfill the missing parts, so as to focus only on our project content. Fortunately, that was not a problem. Redact's vast ecosystem got this covered. For example, there was react-router for route handling or Redux for storage. The most relevant libraries we are using on our client-side application are:

Babel: A few users coming to our site might be using old browser versions, which have little to no support to ES6/ES2017 features. To make sure all browsers can understand our code we make use of Babel, which transpires our modern JavaScript code into JavaScript code that most browsers can understand.

Redux: An in-memory storage for JavaScript. It saves application states, which in other terms are the different data that our application uses over the time. A storage like Redux avoids having to transfer data up and down the React tree, since Redux stores it all in one place which can be accessed anytime. It is also modular, which makes it ideal for our application since it helps towards scalability.
That does not mean that it makes properties and functions we explained earlier become redundant. We should still use these for simple or very specific interactions with components. Nonetheless, Redux simplifies things when working especially with global variables, such as the currently authenticated user.

Redux was initially built for React, so it works hand to hand with it. The storage can be easily connected to React components, which will have access to any of the stored data and also be able to dispatch new actions to add/update the data in it.

3.4.3 Version Control

A version control system can be useful to developers, even when working alone. It enables us to go back in time to figure out what broke a certain utility, work on different features at the time and revert/merge them into the original source code with no difficulty, watch how the project evolved over the time, and so on.

We chose Git. Not only because it is the most popular and widely used version control system, but also because part of our project was the integration with GitHub, and GitHub works with Git.
For the same reason as above, we chose GitHub to be our remote source code repository. Currently, it is a public space where developers can come and have a look at the source code that is powering the chat application, report bugs they encounter or even contribute by submitting pull requests.

Continuous Integration services are automated software that runs as soon as a new version is pushed onto a repository and give the repository contributors constant insight reviews that they would often not do by themselves after pushing every new version.
Given the popularity of GitHub, it counts with many different services that have integrated with it: Continuous Integration services (Travis, CircleCI, Appveyor), Dependency Checkers (David-DM, Greenkeeper, Dependency CI), Code Quality checkers (CodeClimate, Codacy), Code Coverage (Codecov, Coveralls) and many more.

3.4.4 Databases and Models

A key defining aspect of any database-dependent application is its database structure. The database design can vary depending on many different factors, such as the number of reads overwrites or the values that the user is likely to request the most. That is because as full stack developers we want the database to have the best performance, which can often be achieved by focusing the optimizations on the most common actions. We concentrated on the MongoDB database, which is the most complex data storage and the one which stores the most data.

Our Redis data structure limits to mapping sessions to user identifiers, both of type text. That is how a web request works: Node.js queries Redis by using the user session identifier to determine whether the user is signed and their account identifier. If an account identifier is found, Node.js queries MongoDB to find out the rest of the user information.

The MongoDB database stores everything else: users' information, rooms, chats, and messages. Our final database design ended up having four different collections: users, rooms, chats, and messages. Although MongoDB is schema-less, by using the Mongoose library on Node.js, we were also able to define a flexible schema for each of the collections. A schema constrains the contents of a collection to a known format, saving us from validating the structure of the data before or after it has been put in into the database.

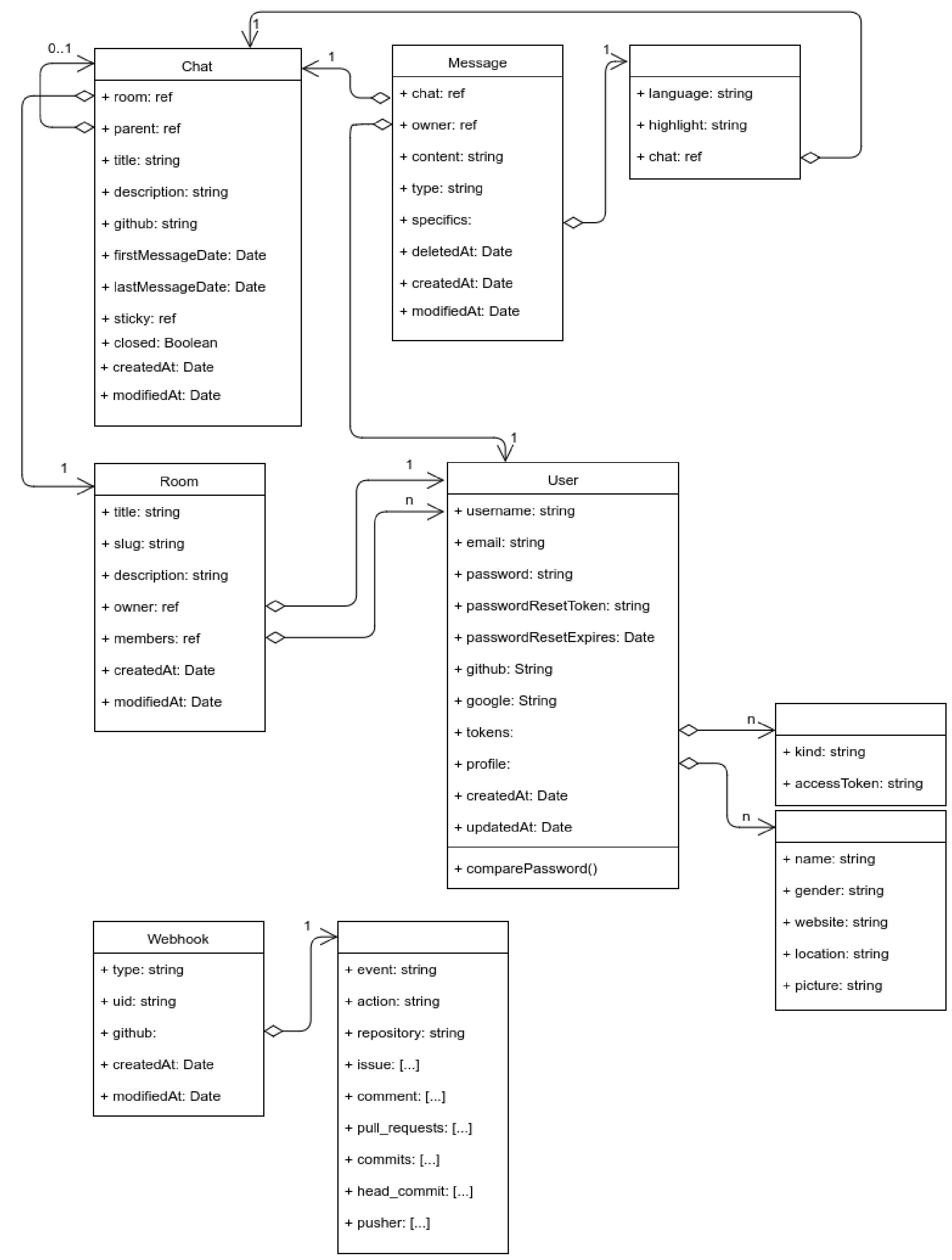


Figure 3Final MongoDB database UML diagram.

These cover most searches, which is what is being done the most often: users are being looked up many times whereas they barely change during their lifetime.

For example, we are searching for the associated user through the _id field on every request, but we only set the _id on their creation. Moreover, we are referring to the email, GitHub, and google identifiers every time a user logs in through each respective method, yet most times these identifiers are only set once during the user's lifespan. Although we did not specify, some of the schema fields are required, whereas others can be left undefined. All these specifications, including each of the fields' validation, were given to Mongoose, either in the form of configuration or functions.

3.4.5. Setting up the development environments

Node

Before getting into Node's configuration itself, we have to stop and think how we are going to share the various data with the client. Given that the client app is not part of the back end whatsoever, our data has to be serialized, preferably into a widely supported format. Traditionally, XML would have been the way to go, but we chose JSON to be our data transport format for the following reasons:

- We are dealing with JavaScript all the time, and JavaScript object structures are very similar to JSON, and can be read and converted to effortlessly.
- JSON is gaining more and more popularity in the web context.
- Express can work with JSON out of the box.
- Companies like Oracle recommend using JSON over XML.

Often, we would use that opportunity to create a RESTful API but given that the back end purpose was only to exchange data with our client, we realized that there was no reason to, at least not a 100% RESTful compliant one.

The reason why we chose not to do that was mainly to speed up development. A RESTful API displays a considerable amount of unnecessary information to anyone who has access to the implementation or extensive documentation of the back end endpoints.

As a result, some of our URL's lack of some CRUD operations, responses do not include linkers such as self or next, error response status are very limited (they are mostly 400 or 500), and so on. Building our back end core was our next step, even though we started with a very basic implementation which eventually got big and complex. As we mention in the technologies chapter, we have decided to use the Express framework, so all our core is built around it. It was easier to configure than expect, and that is because unlike Express 3, Express 4 supports middleware, pieces of code that can be plugged in into the framework to enhance it.

Express is now in charge of managing user sessions, exchanging information with our databases and processing HTTP requests (parsing data such as x-www-form- URL encoded into JavaScript objects, executing certain controllers when a route is matched and returning a value).

CRUD as GET, POST, PATCH, DELETE

GET, POST, PUT/PATCH, DELETE requests are the core of RESTful APIs, and so they are in our application. Although we have previously mentioned that we have no intention to build a complete RESTful back end, following its design patterns can make it easier towards development than trying to build our custom one. It will also make it more understandable by someone who tries to figure out what our application does.

We used GET to retrieve a list of documents or a certain document. Our URLs generally make it pretty obvious to distinct, and most of the times that we require an identifier is because we are looking for a specific one (i.e. /rooms/:slug).

POST is used to create documents. Other sites like Instagram make use of this method to update and delete data as well, most likely for compatibility reasons. We have already extent these requests can be executed by most on the client-side by including fetch polyfills.

DELETE is used to delete a document.

Then there is PATCH, which is used to update documents. You might wonder why we went for PATCH instead of PUT. While both are meant to be used to update documents, the first can update just a few fields while the other is meant to send   the whole new document.

Although PUT is simpler, which we will talk about in a moment, being able to update only a few fields of the model is very interesting when updating chat information. In a chat full of people we are expecting lots of updates to it, and some might come concurrently, that means that the chances to replace the previous user's work are high. If we could reduce that risk to a field at a time, we can, for example, avoid a title to be returned to the original if the user just wanted to modify the description.

React

Now it's time for the React-based client. Although, it is not just about React, but rather React and part of its ecosystem. As we stated earlier, React by itself offers too few utilities to handle a project of this size. There is the need to record application states that will be read and written by other parts of the application, handle URLs and path changes, fetch from the API (with both HTTP and WebSocket's), etc.

Since we fully know the API that we have to support, we can structure our project in a way that prioritizes our back end specifications, and later on, parts of our code will also be able to stick to that rule.

3.4.5 Authentication

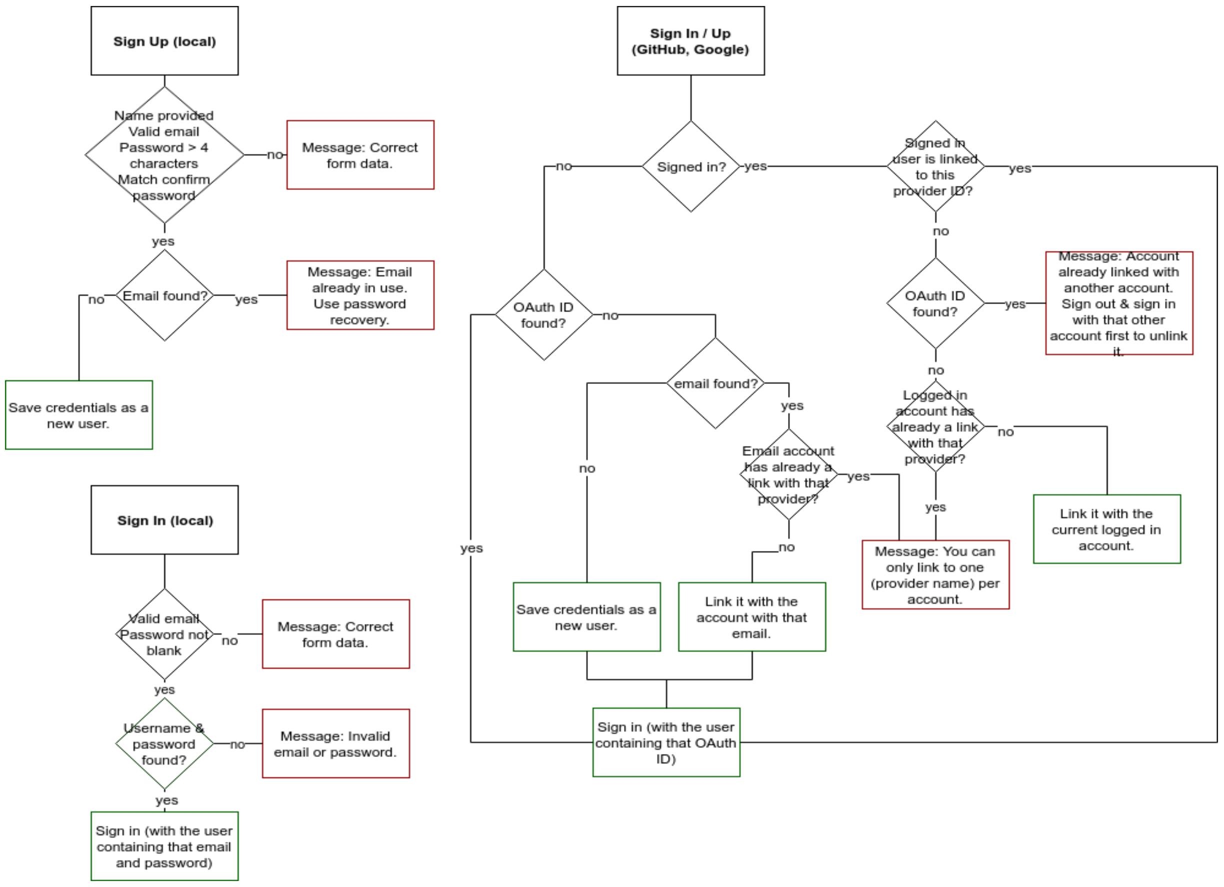
Authentication was our first feature to implement. We wanted to give support to local, GitHub and Google authentications.

On the server-side, this implied creating a few new routes to handle sign in, sign up (only for local authentication) and sign out, and the appropriate strategies to handle each of these providers.

For the local authentication we configured the following two routes:
/auth/signup
/auth/signin

They handle the register and the login form data respectively. For the OAuth authentications, we have these other ones:
/auth/github
/auth/github/callback
/auth/google
/auth/google/callback

We have to have two endpoints for each OAuth authentication. The first one is the request one, which will bring the user to the provider's authentication page, and the callback one is the return URL which the provider will bring the user to after having completed the authentication along with authentication tokens.
It is to note that since we are handling all authentication server-side, even the callbacks, we do not return JSON in any of the AOuth routes. As an exception, we make use of redirects to client pages both when the authentication succeeded and when it errored (either because of a problem on our side or because the user declined to grant us permissions on the provider's page).



Sign Up (local)
Name provided Valid email Password > 4 characters Match confirm password
no
Message: Correct form data.
yes
Email found?
yes
Message: Email already in use. Use password recovery.
no
Save credentials as a new user.
Sign In (local)
Valid email Password not blank
no
Message: Correct form data.
yes
Username & password found?
no
Message: Invalid email or password.
yes
Sign in (with the user containing that email and password)
Sign In / Up (GitHub, Google)
Signed in?
no
yes
Signed in user is linked to this provider ID?
yes
no
OAuth ID found?
yes
Message: Account already linked with another account. Sign out & sign in with that other account first to unlink it.
no
Logged in account has already a link with that provider?
no
Link it with the current logged in account.
yes
Message: You can only link to one (provider name) per account.
OAuth ID found?
no
yes
email found?
no
yes
Email account has already a link with that provider?
yes
no
Save credentials as a new user.
Link it with the account with that email.
Sign in (with the user containing that OAuth ID)

Figure 4:Flow Dragram

Back end tests consist of making sure that the API correctly handles, stores and returns JSON responses to the user (through AJAX or WebSocket's). Our express architecture is divided into 4 parts:

1) Models
2) Services,
3) Controllers
4) Router.

Models

Mongoose Models are the life and soul of our application. They define our MongoDB structure and enforces the types of data it will support. Thanks to Mongoose providing us this layer above the database and making it completely isolated from the database implementation, unit testing models is not a big deal. A Mongoose Schema (which is the definition of a Model) can be divided into 4 parts: data types, data validators, middleware (which can be attached before or after saving/updating data) and methods.

Although we can test the Schema right away by using integration tests, it is often preferable to test each of the parts individually. For this reason, we should try to use integration tests only to make sure that they all work well together.

Services

We can describe services as logic that often operates with Model(s) data. When it requires of Models' data, we can use stubs in order to provide a predictable database response, which will make it fully unitary too. The other, not very common case, are services that do the work by their own. Their response is in most cases based on the parameters given. In this case, unit testing them is trivial; all we have to do is make sure that their response for certain given parameters is the one we are expecting.

Controllers

Controllers duty is to process router's input and call the appropriate services, by making the appropriate transformations to that input. Generally, they are very simple, and not really worth writing unit tests for them. To unit test them, we can do exactly what we did with services, with the exception that in this case the stubs will be made over service functions other than Model methods.

Router

Our MVC structure allowed us to make our router extremely simple, to the extent that nothing but the two router handlers were worth testing. Our router routes looked like this:
router.get('/users/whoami', c(user.whoami, req=>[req.user]));
We could have also tested this by making sure that the controller function was really called whenever a specific get, post, put or delete were called, but we thought it was best investing our effort elsewhere, especially because integration + E2E tests were already covering this in most of the tests.

Client

In order to handle the local sign in and sign up, we created two different forms (figures 4.3 and 4.4 respectively) which will send their information to the server on submit. The server will verify them and respond the client with any error that might have occurred.

Similarly, there is a button with no fields associated that starts the sign-out process. When using Redux notifying different parts of the application is straightforward, and authentication is probably one of the best use cases for it. We can update the Redux storage from any component at any time, and all other components listening to Redux updates will get the changes immediately. For example, once the user has just signed in from the local authentication form, we can have the navbar updated with their name without any sort of timers nor messy tree properties sharing. Below there is a diagram of what our Authentication Redux module looks like.

Features

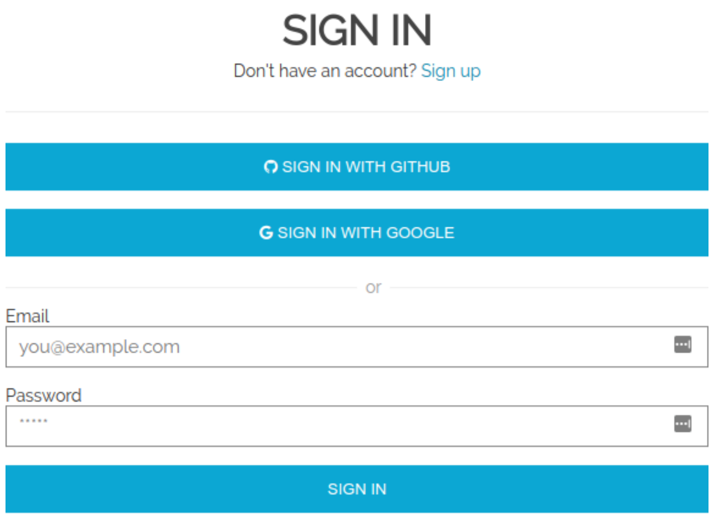


Figure 5 Sign in form

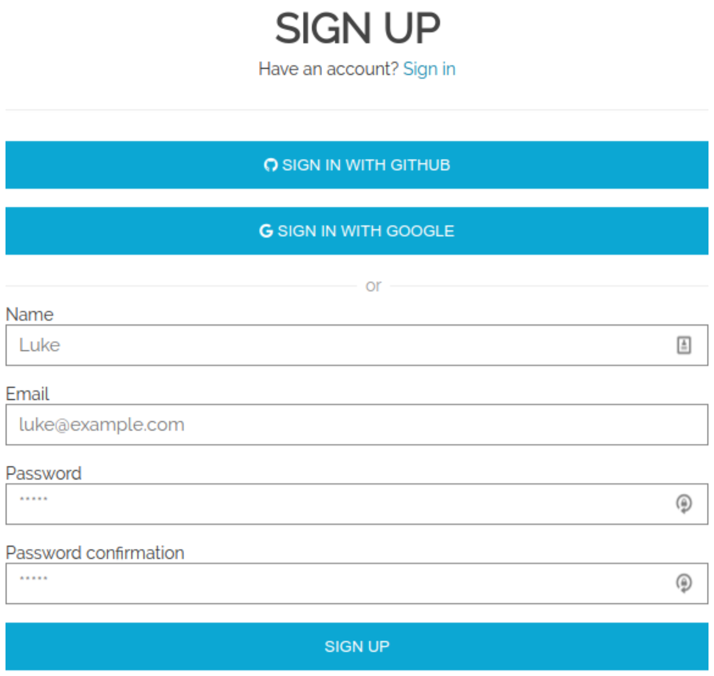


Figure 6:Sign up form

UML Diagram:

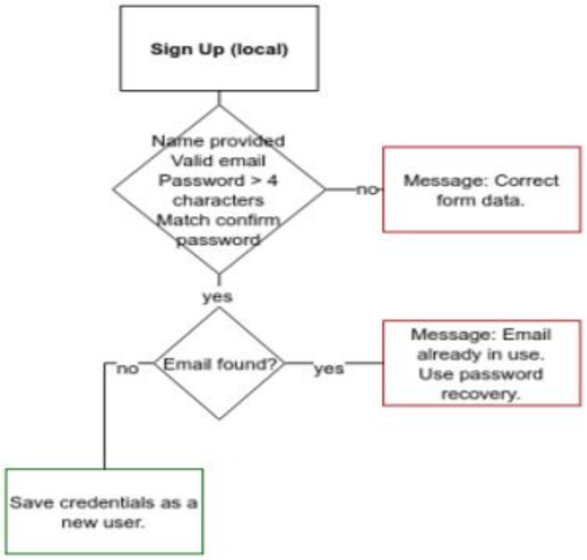

Sign Up (local)
Name provided Valid email Password > 4 characters Match confirm password
no
Message: Correct form data.
yes
Email found?
no
yes
Message: Email already in use. Use password recovery.
Save credentials as a new user.

Figure 7:signup

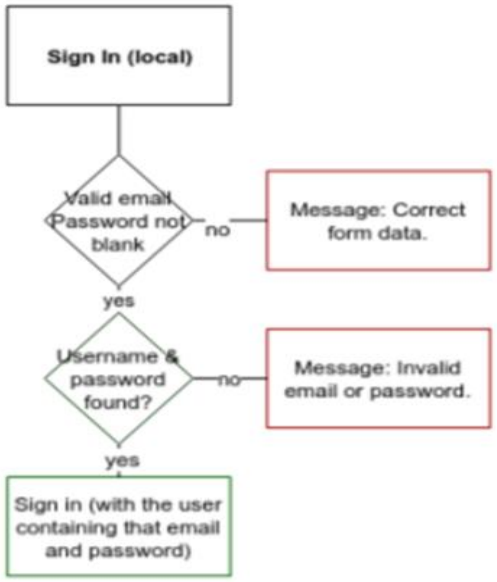

Sign In (local)
Valid email Password not blank
no
Message: Correct form data.
yes
Username & password found?
no
Message: Invalid email or password.
yes
Sign in (with the user containing that email and password)

Figure 8:signup

Since Redux is just an in-memory storage, it will be empty every time to a user enters our page. To fetch all their user data once again, we will make use of the users/whoami API route to refill our Redux storage once our application boots.

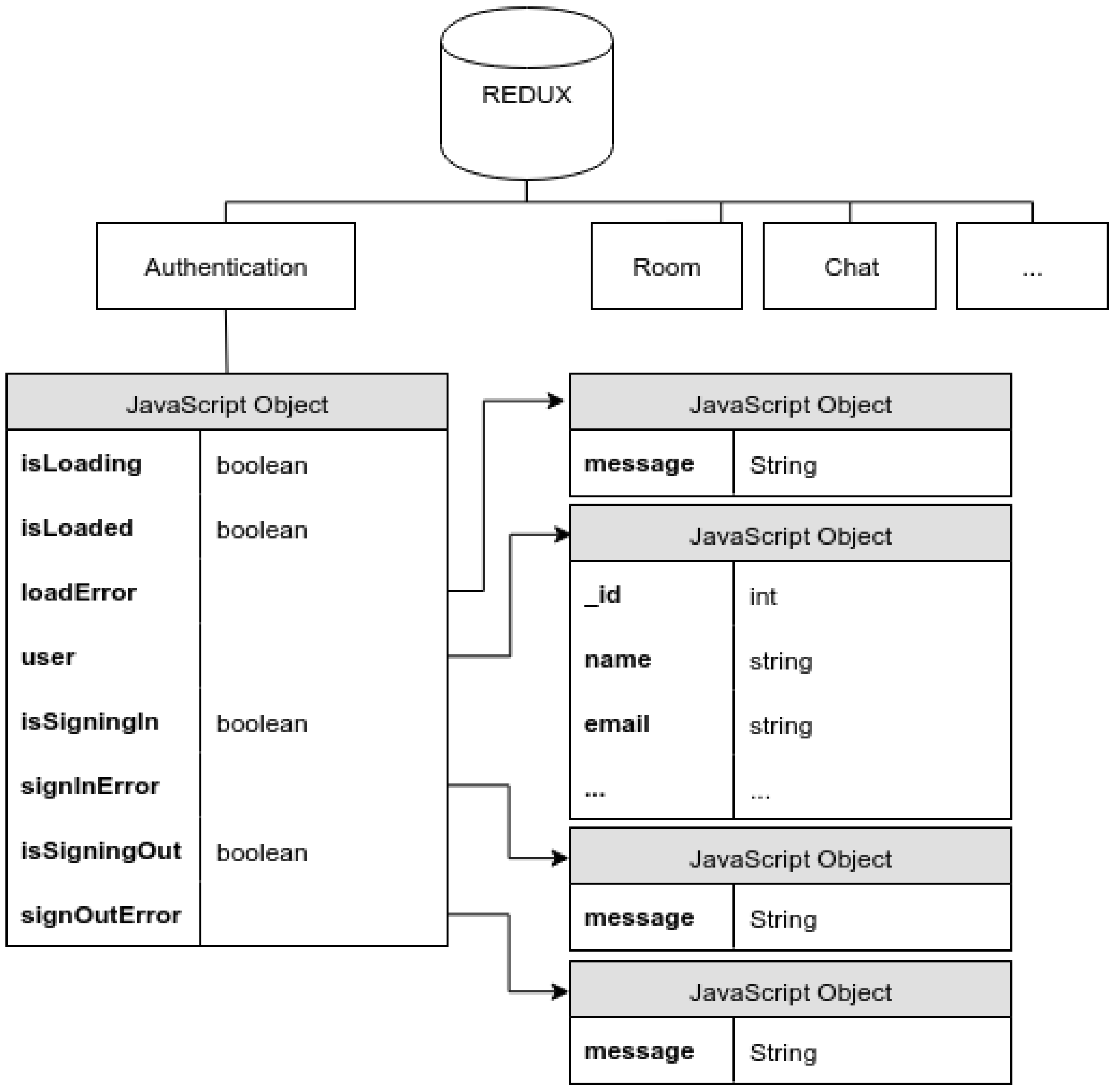


Figure 9:Diagram of Redux authentication module

API

GitHub and Google logins require a redirection to their sites (such as the one shown on figure 4.6), for the reasons we just stated when describing the server duties. Since the server does all this work for us, all we have to is to redirect to the server endpoint in charge of starting the OAuth authentication and recheck the user state when the user comes back (like if the user was returning to our site). If the user now appears to be signed in the authentication succeeded, otherwise we display the error message sent as query parameters from the server.

CRUD Operation Table

Once the authentication part was ready, it was time to move on onto the Rooms, Chats, and Messages that users will Create, Read, Update and Delete (CRUD).

It was an easy but tedious process. Each operation involved database operations, a client to server request, specific validation, UI (a form or grids of data), and displaying the result of each operation back to the user when it was complete.

On the server side, we are supporting these operations by using GET, POST, PATCH and DELETE method will be explained further.

To do so effectively, we started abstracting the most common operations, especially validation ones (which we are also going to reuse later): verify whether the user is logged in, ownership, existing rooms, chats and messages, etc. We do try to follow the Do Not Repeat Yourself (DRY) principle.

The API endpoints that take part in each of the topics are the following:

Rooms:

Method	Route	Description
GET	/rooms	List of rooms
POST	/rooms	Create a new room
GET	/rooms/search	Filter rooms by providing optional query parameters: _id, slug or title.
GET	/rooms/_id	Get a specific room given an existing room identifier.
PATCH	/rooms/:_id	Update a specific room given an existing room identifier and valid room fields.
POST	/rooms/:_id/join	Join a specific room given an existing room identifier. A signed in user is required.
POST	/rooms/:_id/leave	Leave a specific room provided a valid room identifier. A signed in user is required.
DELETE	/rooms/:_id	Delete a specific room given an existing room identifier. Requires to be signed in as the room owner.
GET	/rooms/:_id/chats	List of room chats given an existing room.
POST	/rooms/:_id/chats	Create a new room chat given an existing room identifier. Requires to be signed in as the room owner.

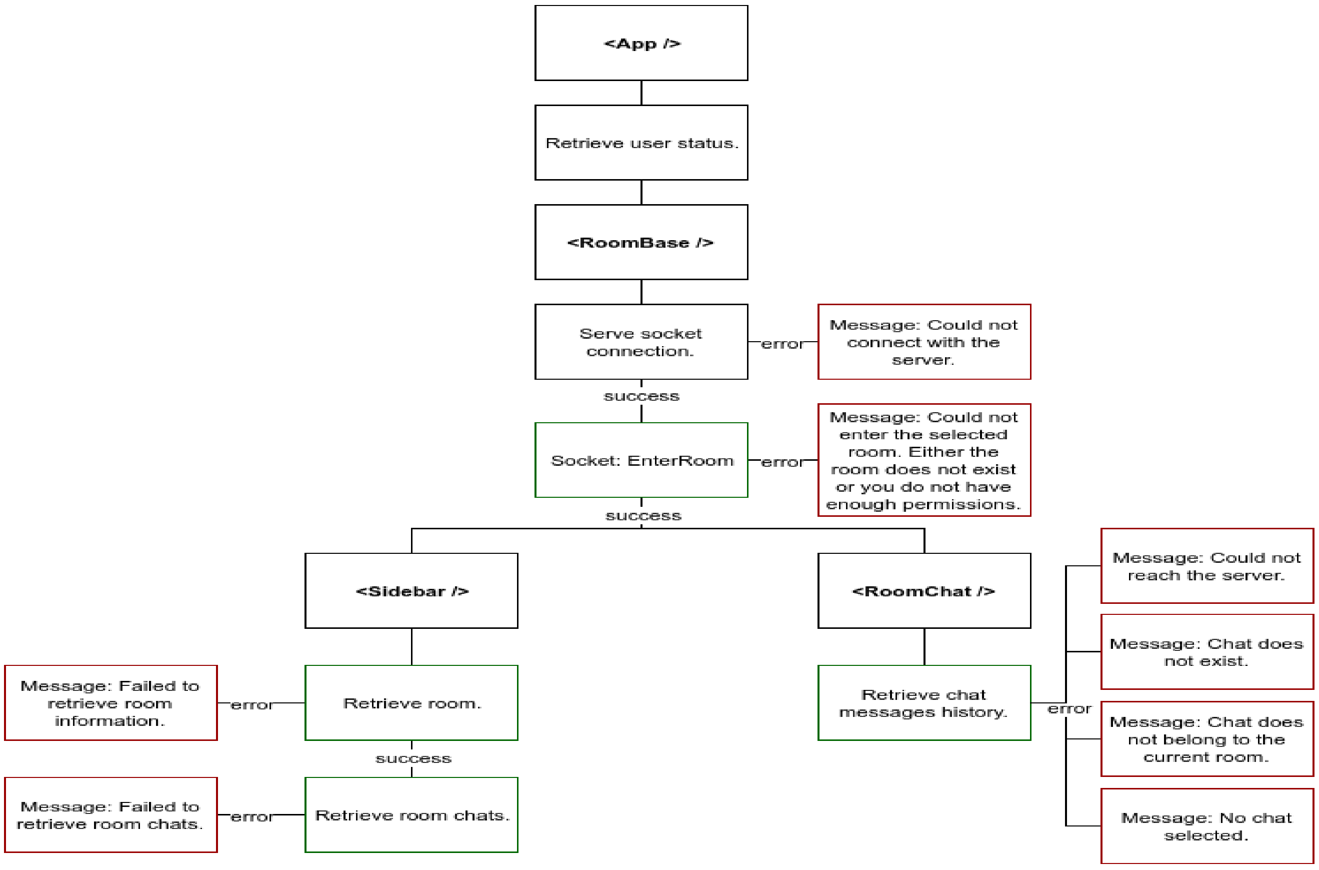
Chats

Method	Route	Description
GET	/chats/:_id	Get a specific chat given an existing chat identifier.
PATCH	/chats/:_id	Update a specific chat given an existing chat identifier and valid chat fields.
DELETE	/chats/:_id	Delete a specific chat given an existing chat identifier.
POST	/chats/:_id/fork	Fork a chat.
POST	/chats/:_id/fork-merge	Merge a fork with the original chat.
POST	/chats/:_id/fork-upgrade	Move fork a new chat.
GET	/chats/:_id/messages	Get chat messages given an existing chat identifier.
POST	/chats/:_id/messages	Create a chat message given an existing chat identifier.

Messages

Method	Route	Description
PATCH	/messages/:_id	Update the content of a message given a message identifier. Requires to be signed in as the message owner.
DELETE	/messages/:_id	Delete a message given a message identifier. Requires to be signed in as the message owner.

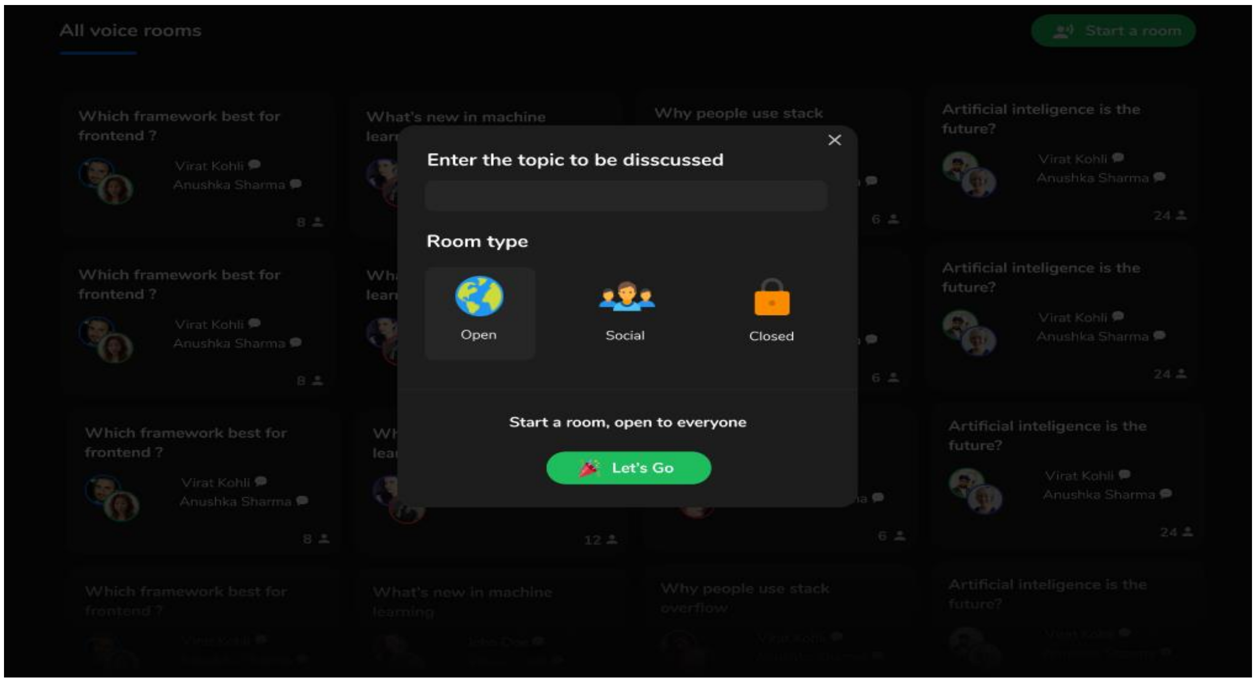
In the previous section, we have described how we are using WebSocket's to send and receive messages in a chat room.



<App />
Retrieve user status.
<RoomBase />
Serve socket connection.
error
Message: Could not connect with the server.
success
Socket: EnterRoom
error
Message: Could not enter the selected room. Either the room does not exist or you do not have enough permissions.
success
<Sidebar />
<RoomChat />
Message: Could not reach the server.
Message: Chat does not exist.
Message: Failed to retrieve room information.
error
Retrieve room.
Retrieve chat messages history.
error
Message: Chat does not belong to the current room.
success
Message: Failed to retrieve room chats.
error
Retrieve room chats.
Message: No chat selected.

Figure 10:React Room activity diagram

Snippets



Enter the topic to be disscussed

Room type

Start a room, open to everyone

Figure 11: new Room

4.2.REQUIREMENT ANALYSIS:

Requirement analysis is for transformation of operational need into software description, software performance parameter, and software configuration through use of standard, iterative process of analysis and trade-off studies for understanding what the customer wants, analyzing need, assessing feasibility, negotiating a reasonable solution, validating the specification and managing the requirements.

GENERAL CONSTRAINTS

1. Hardware Limitations: There are no hardware limitations.
2. Interfaces to other Applications: There shall be no interfaces.
3. Parallel Operations: There are parallel operations.
4. Audit Functions: There shall be no audit functions.
5. Control Functions: There shall be no control functions

4.4.1. FUNCTIONAL REQUIREMENTS:

The relationship between the input and output to the system is determined by the functional requirement of the SRS.

TECHNICAL ISSUES:

Technical issues are a key step while developing a software application. A software project has failed due to an incomplete or inaccurate analysis process, especially technical issues.

RISK ANALYSIS :

Their main challenge is to determine how to model and visualize the complex relationships between risks, define and monitor the risks‘ impacts, analyze the probability of risk occurrence, mitigate the negative impact of risks, and monitor the course of the project with risks and uncertainties.

4.5.2. NON FUNCTIONAL ATTRIBUTES

SECURITY: The project provides security to different kinds of customers by means of authentication level. The authorization mechanism of the system will block the unwanted attempts to the server.

RELIABILITY: The project is guaranteed to provide reliable results for the entire user. The system shall operate 95% of the time.

SCALABILITY: The need for scalability has been a driver for much of the technology innovations of the past few years.

MAINTAINABILITY: Maintainability is our ability to make changes to the product over time. We need strong maintainability in order to retain our early customers.

4.5.3. <u>Iterative Waterfall Model</u>

In a practical software development project, the classical waterfall model is hard to use. So, the Iterative waterfall model can be thought of as incorporating the necessary changes to the classical waterfall model to make it usable in practical software development projects. It is almost the same as the classical waterfall model except some changes are made to increase the efficiency of the software development. The iterative waterfall model provides feedback paths from every phase to its preceding phases, which is the main difference from the classical waterfall model.

When errors are detected at some later phase, these feedback paths allow correcting errors committed by programmers during some phase. The feedback paths allow the phase to be reworked in which errors are committed and these changes are reflected in the later phases. But there is no feedback path to the stage – feasibility study, because once a project has been taken, does not give up the project easily.

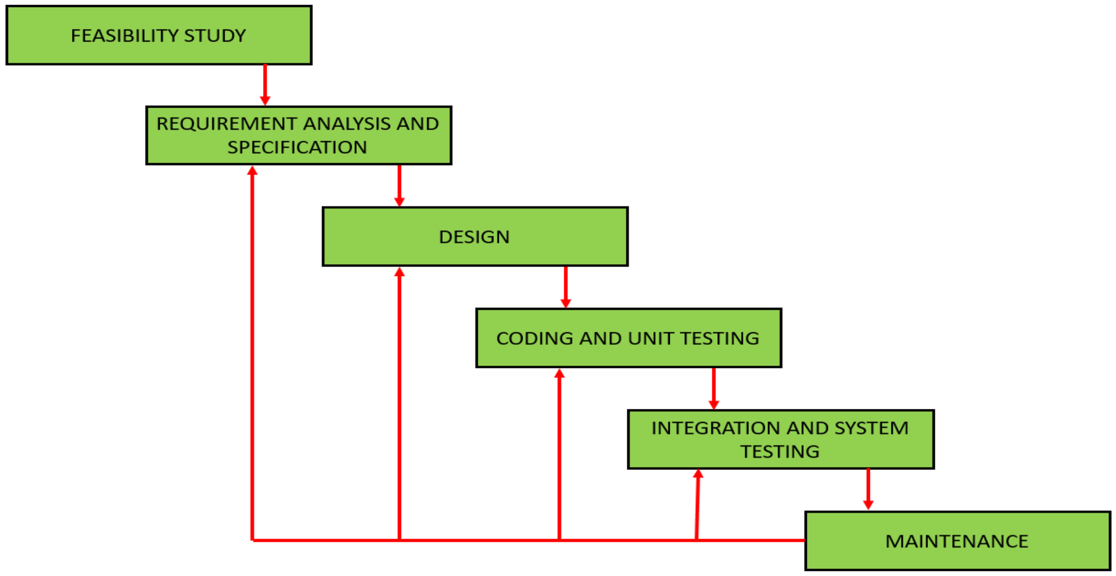

FEASIBILITY STUDY
REQUIREMENT ANALYSIS AND SPECIFICATION
DESIGN
CODING AND UNIT TESTING
INTEGRATION AND SYSTEM TESTING
MAINTENANCE

Figure 12: Iterative Waterfall Model

Phase Containment of Errors :
The principle of detecting errors as close to their points of commitment as possible is known as Phase containment of errors.

Advantages of Iterative Waterfall Model :

Feedback Path –
In the classical waterfall model, there are no feedback paths, so there is no mechanism for error correction. But in the iterative waterfall model feedback path from one phase to its preceding phase allows correcting the errors that are committed and these changes are reflected in the later phases.

Simple –
Iterative waterfall model is very simple to understand and use. That’s why it is one of the most widely used software development models.

Cost-Effective –
It is highly cost-effective to change the plan or requirements in the model. Moreover, it is best suited for agile organizations.

Well-organized –
In this model, less time is consumed on documenting and the team can spend more time on development and designing.

Drawbacks of using this Iterative Waterfall Model :

Difficult to incorporate change requests –
The major drawback of the iterative waterfall model is that all the requirements must be clearly stated before starting the development phase. Customers may change requirements after some time, but the iterative waterfall model does not leave any scope to incorporate change requests that are made after the development phase starts.

Incremental delivery not supported –
In the iterative waterfall model, the full software is completely developed and tested before delivery to the customer. There is no scope for any intermediate delivery. So, customers have to wait a long for getting the software.

Overlapping of phases not supported –

Iterative waterfall model assumes that one phase can start after completion of the previous phase, But in real projects, phases may overlap to reduce the effort and time needed to complete the project.

Risk handling not supported –
Projects may suffer from various types of risks. But, the Iterative waterfall model has no mechanism for risk handling.

Limited customer interactions –
Customer interaction occurs at the start of the project at the time of requirement gathering and at project completion at the time of software delivery. These fewer interactions with the customers may lead to many problems as the finally developed software may differ from the customers' actual requirements.

Chapter-5

Deployment and Conclusion

5.1. Deployment

Since the very beginning, the chat application was meant to be a cloud service. Users would be able to access it anytime without having to install themselves any special software. Thus, we had to upload our working software on a remote server, which was accessible worldwide. Although we marked the deployment as Phase 4, we began deploying our product at the end of Phase 3, when we already had a decent set of utilities, and the platform was already usable through the UI. We used a Digital Ocean VPS server to conduct our deployment tests.

Leaving apart all the operating system and network security details that every system administrator has to take care of when deploying professional cloud services, to have our source code running on a remote server we required a MongoDB and Redis instances running all the time and a Node.js and a React server running with production settings.

Only at that point visitors, given a valid address (domain or IP), would be able to see our site just as if it was running on our development machines.

5.2 Docker

Docker is an open-source platform for developers and system administrators to build, ship, and distribute applications. Docker works with software "containers." A container can store any kind of software with all they need to run dependencies, configuration, and other tools. By using a Docker container, we can run all our source code with its production settings and databases with a single command, and no specific applications but Docker. Soon we realized that Docker provides an even better solution to this: Docker Com- pose.

Docker Compose can manage several Docker containers. Each of them can communicate with each other and have a set of shared configurations while having their runtime dependencies isolated from one and the other.

Conclusion

In this paper, we introduced a specification for preserving the security and privacy of the chat application. We described a set of requirements for making secure chat and implement it by using modern methods and lightweight for providing speed and good protection to its clients. XSalsa20 algorithm ideal for mobile devices because of its high security, high performance and maintains battery life. Clients can be confident that nobody can read their messages, even if the mobile phone reaches wrong hands cannot enter to the application and cannot access the data stored locally.

Reference

- “Webrtc vs websockets.”http://stackoverflow.com/a/18825175, sep 2013. Accessed: 2017-04-19.
- “Webrtc samples.” https://webrtc.github.io/samples/, sep 2013. Accessed: 2017-04-19.
- “User stories: An agile introduction.”http://www.agilemodeling.com/ artifacts/userStory.htm. Accessed: 2016-10-20.
- “Making the switch from making the switch from node.js to golang. ”http://blog.digg.com/post/141552444676/ making-the-switch-from-nodejs-to-golang, mar 2016. Accessed: 2016-10-19.
- “Sinon-best practices for spies, stubs and mocks.” https://semaphoreci.com/community/tutorials/best-practices-for-spies-stubs- and-mocks-in-sinon-js, 2016. Accessed: 2016-10-03.
- “Why is nosql faster than sql?.” http://softwareengineering. stackexchange.com/questions/175542/why-is-nosql-faster-than-sql, nov 2012. Accessed: 2016-10-21.
- “Why nosql.”http://softwareengineering.stackexchange.com/questions/175542/why-is-nosql-faster-than-sql. Accessed:2016-10-21.
- “Exploring the different types of nosql databases.” https://www. 3pillarglobal.com/insights/exploring-the-different-types-of-nosql-databases, may 2015. Accessed: 2017-04-29.
- “Composition vs inheritance - react.” https://facebook.github.io/react/ docs/composition-vs-inheritance.html. Accessed: 2016-12-18.
- “Mongodb manual 3.4.” https://docs.mongodb.com/manual/reference/ limits/. Accessed: 2017-08-30.
- “6 rules of thumb for mongodb schema de- sign.” http://blog.mongodb.org/post/87200945828/ 6-rules-of-thumb-for-mongodb-schema-design-part-1, may 2014
- Accessed: 2016-10-18. “Scaling secret: Real-time chat.” https://medium.com/always-be-coding/ scaling-secret-real-time-chat-d8589f8f0c9b#.m5jigxq6x, may 2015.
- Accessed: 2016-10-18. “Analysis of json use cases.” https://blogs.oracle.com/xmlorb/entry/ analysis_of_json_use_cases, apr 2013. Accessed: 2016-11-08.
- Crud cycle (create, read, update and delete cycle).” http:// searchdatamanagement.techtarget.com/definition/CRUD-cycle. Ac- cessed: 2016-11-08.
- “About native xmlhttp.” https://msdn.microsoft.com/en-us/library/ ms537505(v=vs.85).aspx. Accessed: 2016-12-18.
- “Please. don’t patch like an idiot..” http://williamdurand.fr/2014/02/14/ please-do-not-patch-like-an-idiot/, aug 2016. Accessed: 2016-12-18.
- “Rfc 5789 - patch method for http.” https://tools.ietf.org/html/rfc5789, mar 2010. Accessed:

2016-12-18.

- “Perfomance tips | google cloud platform.” https://cloud.google.com/ storage/docs/json_api/v1/how-tos/performance#patch. Accessed: 2016-12-18.
- “container vs component?.” https://github.com/reactjs/redux/issues/ 756#issuecomment-141683834, sep 2015. Accessed: 2016-11-13.
- D. T. Andrew Hunt, The Pragmatic Programmer: From Journeyman to Master.
- Addison-Wesley Professional, first ed., october 1999. “Html5 websocket: A quantum leap in scalability for the web.” http://www. websocket.org/quantum.html. Accessed: 2016-11-13.
- “Building a scalable node.js express app.” https://medium.com/@zurfyx/ building-a-scalable-node-js-express-app-1be1a7134cfd, feb 2017. Ac- cessed: 2017-04-13.
- “Strategy design pattern.” https://sourcemaking.com/design_patterns/ strategy. Accessed: 2016-11-14.
- “Dynamic testing.” https://www.tutorialspoint.com/software_testing_ dictionary/dynamic_testing.htm, apr 2017. Accessed: 2017-04-03.