

**DEPARTMENT OF COMPUTER ENGINEERING**

**JSPM’s IMPERIAL COLLEGE OF ENGINEERING & RESEARCH WAGHOLI, PUNE- 412207**

**2022-23**

**“Online chatting app:-Whatsappclone”**

DBMS MINI PROJECT

***Submitted to***

**SAVITRIBAI PHULE PUNE UNIVERSITY, PUNE**

***Submitted By***

1)Sejal Khalate -TE2126

2)Shweta Medhe -TE2131

Under the guidance of – Mr. N.Bansode



**DEPARTMENT OF COMPUTER ENGINEERING**

**JSPM’s IMPERIAL COLLEGE OF ENGINEERING & RESEARCH WAGHOLI, PUNE- 412207**

**2022-23**

This is to certify that the DBMS MINI PROJECT entitled “ Online Chatting App:- Whatsappclone“ submitted by 1)Sejal Khalate 2)Shweta Medhe is a record of the bonafide work carried out by her, under my guidance as a part of DBMS MINI PROJECT..

**Guide Mr.N.Bansode Head of Department** Dr.V.Wadne Principal Dr.R.Deshpande

**(External Examiner)**

# ABSTRACT

The purpose of Online Chat application is to automate the existing manual system by the help of computerized equipment and full fledge computer software, fulfilling their requirement, so that their valuable data/information can be stored for a longer period with easy accessing and manipulating of the same . The required software and hardware are easily available and easy to work with.

Online Chat Application , as described above, can lead to error free, secure, reliable and fast management system. It can assist the user to concentrate on their other activities rather to concentrate on the record keeping. Thus it will help organization in better utilization of resources. That means that one need not to be distracted by information that is not relevant, while being able to reach the information.

The aim is to automate its existing manual system by the help of computerized equipment and full-fledge computer software, fulfilling their requirement, so that their valuable data/information can be stored for a longer period with easy accessing and manipulating of the same. Basically the project describes how to manage for good performance and better service for the users. i  **ACKNOWLEDGMENT**

First and foremost, I would like to express my most sincere gratitude to my professor, Ashish Gaigol Sir, for its constant support and knowledge throughout the course of the project. Without his help, I am sure the project quality would have been lower, especially when it comes to optimization and code design. You have been able to resolve my most complex concerns and questions that have arisen during all this time, and provided me with plenty of documentation to widen my knowledge. Thank you, Neat Roots, for all you have taught me about App development and helping me with the deployment. Also, my thanks to my classmates at the JSPM’s ICOER, for their time and valuable feedback about the application. Thank you Sayali Pawar for going through my app and providing insightful reports. Thank you, my groupmates for all the feedback you have given me about the code itself, enhancements and best practices, primarily when it comes to React. Lastly, I would like to thank my parents and brother for their time and support during all this time.

**ii**

**CONTENTS**

**Abstract** …………………………………………………………………… i

### **Acknowledgements** ………………………………………………….ii

1. **Introduction 3** 
   1. Aim . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .1
      1. Features . . . . . . . . . . . . . . . . . . . . . . . . . .1
2. **Phase 1: Research** ……………………………………………………..2
   1. Competitive Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . 3
   2. The Ideal Platform . . . . . . . . . . . . . . . . . . . . . . . 4
   3. Communication Protocols . . . . . . . . . . . . . . . . . . . . . . 5
3. **Phase 2: Planning, Prototypes, and Technology** ………….7
   1. Project Methodology . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
   2. Use Cases and Scenarios . . . . . . . . . . . . . . . . . . . . . . . . 7
   3. Estimated Timeline . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
   4. Technology . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
      1. Back End . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
      2. Front End . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
4. **Phase 3: Implementation** …………………………………………….13
   1. Databases and Models . . . . . . . . . . . . . . . . . . . . . . . . . . 13
      1. Users . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
      2. Rooms . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
      3. Chats . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

iii **Chapter 1**

**Introduction**

Aim 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 developers 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.

**1.1.1 Features**

Before getting into any specific chat features that our application should/could have, we will list the basic ones that most chat services offer us nowadays, regardless of their type: • Instant messaging

* Notifications
* Message sender (username and avatar)
* Status
* Group chats
* Audio notes
* Voice calls

When it comes to programming features, the ones we were the most interested in, we have split them into three categories: technical, code and github.

**Technical**

* app development based on java
* Sticky/pinned messages

1

* Discussion forking (based on a previous message)
* Real-time polls
* Display in-line URL description
* Reputation/points system
* Public API
* Bots integration

**Code**

* Code formatting
* Code highlighting
* Edit previously shared code (i.e. to share an improvement) — like in a version control system
* Display in-line activities and fragements results

**GitHub**

* Accessing required dependencies.
* Specific repository chats
  + - Watch commits
    - Watch and discuss issues and pull requests
    - Sync issue discussion with GitHub

2

**Chapter 2**

Phase 1: Research

**2.1 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 indepth 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.

During the competitive analysis, we investigated the following platforms:

* Flowdock - [https://www.flowdock.com](https://www.flowdock.com/)
* Gitter - [https://gitter.im](https://gitter.im/)
* Hangouts - [https://hangouts.google.com](https://hangouts.google.com/)
* Matrix - [http://matrix.org](http://matrix.org/)
* Messenger - [https://messenger.com](https://messenger.com/)
* Rocket.chat - [https://rocket.chat](https://rocket.chat/)
* Skype - [https://web.skype.com](https://web.skype.com/)
* Slack - [https://slack.com](https://slack.com/)
* Telegram - [https://web.telegram.org](https://web.telegram.org/)

3

* Whatsapp - [https://web.whatsapp.com](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.

We are also planning to have some other unique features when it comes to sharing code, such as the possibility to visualize code results straight from activies and fragments.

**2.2 The Ideal Platform**

A term which will be referred to throughout the document is Ideal platform. The Ideal platform represents the ideal chat application and the greatest user experience a developer can have when utilizing our application.

It comes as a result of the competitive analysis research.

4

Since it has a plethora of features, which are an impossibility to tackle as a solo developer, it has just served as a guideline throughout the development. As we describe in the "Phase 2" section, we have used Agile methodology to prioritize all this work and get the maximum amount of relevant pieces working.

The Ideal platform’s UX could be described as follows:

Newcomers can identify themselves by either creating a new account (local authentication) or by authenticating through a social account, such as Facebook, Google or Twitter.

The site will encourage the visitor to authenticate through GitHub, since granting GitHub permissions enables a few developer specific tools. This includes displaying their repositories list, issues, forks, pull requests and commits.

Choosing a unique username is a requirement. We will be using the member’s chosen username as a visual identification on the various utilities, such as group conversations or profile pages, instead of their real name.

A registered user can create their own rooms, which can be public or private. A "room" is nothing but a container for several "chats" (where the conversations happen). Project administrators might want to create their own rooms for each of their projects, having one or more chats to discuss its development. Having more than one chat is useful to divide the different topics inside a project, instead of having conversations of mixed contents altogether.

Any user can read and join other member’s rooms if these are public or one of their members has invited them. A room administrator can create and remove its chats. Chats can be either generic or git based. Members should choose the last one if they have a remote source code repository, such as GitHub. Git based chats have a tight integration with the remote git provider platform, such the ability to watch commits or issues in real time.

Rooms can be bookmarked, which makes it straightforward to access them later from the homepage, instead of having to look them up on the site’s search engine or access them through a direct link.

Furthermore, there is a room-based reputation system to promote group discussions and productivity. Users can get reputation by either chatting or working (as GitHub activity counts towards reputation).

When it comes to chatting features, there are many. A few of them are very well known, such as file sharing. Others, in part, because they are very specific not so much, such as code sharing or MarkDown formatting. In the following section, we will go into details about them.

#### 2.3 Communication Protocols 5

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, WebSockets, 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.

WebSockets are a better approach. WebSockets 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[1]. 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 WebSockets). The fact that stream data can be exchanged between clients directly often means faster messaging and less server-side workload.

**6**

**Chapter 3**

# Phase 2: Planning, Prototypes, and Technology

**3.1 Project Methodology**

Android studio is a set of techniques to manage app 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 android studio, we were able to focus only on the features which had the most priority at the time.

## 3.2 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.

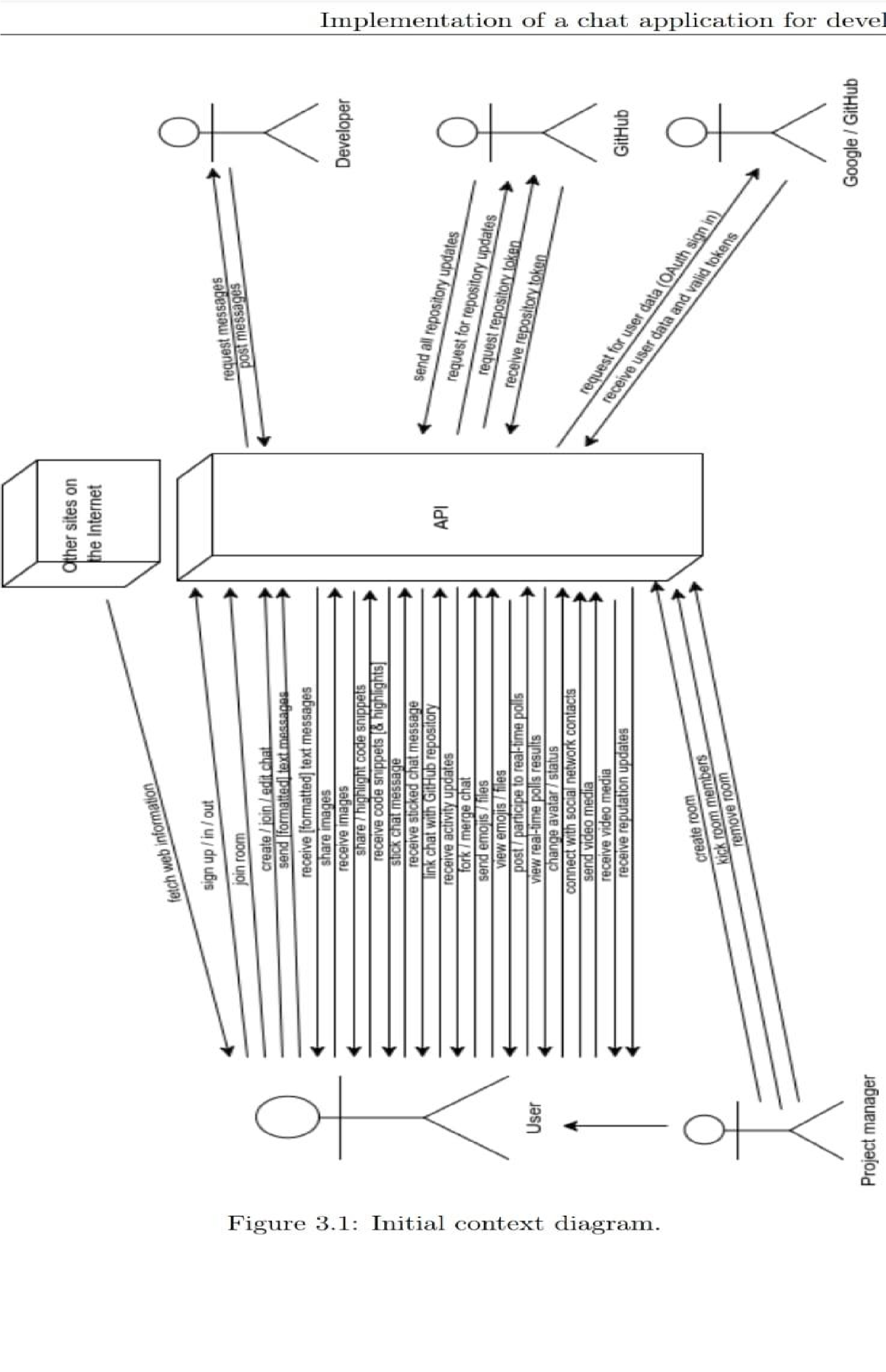
Since we were working with Agile, this list did not have to be complete before we started working on the project, but it was desirable to have at least a few items to start with so that we could establish proper feature priorities.

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.

As you might have noticed already, product backlog, and consequently sprint back-logs too, go hand to hand with the features described in our Ideal platform. It makessense that we have plenty of user stories since they describe the characteristics thatwould be desirable to have, 7 that also means that a few of them will remainundone. We were able to complete all user stories which had a priority of five and above.

8



## 3.3 Estimated Timeline

Based on the initial user stories, we wrote the following timeline.

We built it taking into consideration the product backlog priorities and each of the feature’s development time to make an approximation of what could be done within a semester.

It did not mean to be our final timeline, and we were expecting some features to be developed sooner or later than anticipated, and to have few new user stories every sprint.

Week Tasks

0-4week task:- Initial report with app objective, time schedules, and other valuable starting data.

Build application boilerplate which will serve as the base of our app.

1. week task:-User registration and authentication.
2. week task:- Implementation of public rooms, which users will

be able to join freely.

7week task:- Development of a basic group chat inside a room so that users can communicate to each other. It will cover the following features: Instant messaging, group chats.

8week task:-Desktop and site notifications, user status (online, busy, away), ability to type and send emojis, MD and limited HTML formatting.

9week task:- Implementation of room roles.

10week task:-Ability to stick messages, highlight code, improve previously sent code, and proper code formatting depending on the language it has been typed in.

11 week task:- Documentation (handling mid-term report the following week). Review/finish parts of current logic and design that might have been postponed.

11 week task:- Dedicated GitHub repository chat: watch commits, issues, and pull requests. 12 week task:-Discussions forking. Users will be able to fork a chat based on a previous comment/item, issue, commit, pull request.

1. week task:- Reputation system based on users activity (on the platform and on GitHub), such as pushing commits or creating new issues.
2. week task:- Private rooms, including administrator(s) invitations to join a room.
3. week task:- Deployment of the API, database, and client on a cloud server.

## 3.4 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 environment 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 unquestionably 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).

##### 3.4.2 Front End

Having separated the server-side from the client side, a SPA (Single-Page Application) 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 cases.

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.

**Chapter 4**

# Phase 3: Implementation

This chapter details the most relevant parts of the application development, decisions taken and algorithms.

We have divided this chapter into three sections: databases (design), features (the most important ones) and a brief overview on how we tested our features.

#### 4.1 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 over writes 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.

###### 4.1.1 Users

To start, we needed somewhere to store our users. Since we were expecting a significant number of entries, an individual collection for the users’ themselves was the most appropriate.

What we mean by that is that it was best for the users’ collection to solely store the

13

information that made reference to their authentication and personal data. Their rooms, chats, and messages should be stored somewhere else.

Given that we were expecting a lot of rooms, chats, and messages per user, we refrained from even making references to them in this collection. We are querying these other collections directly.

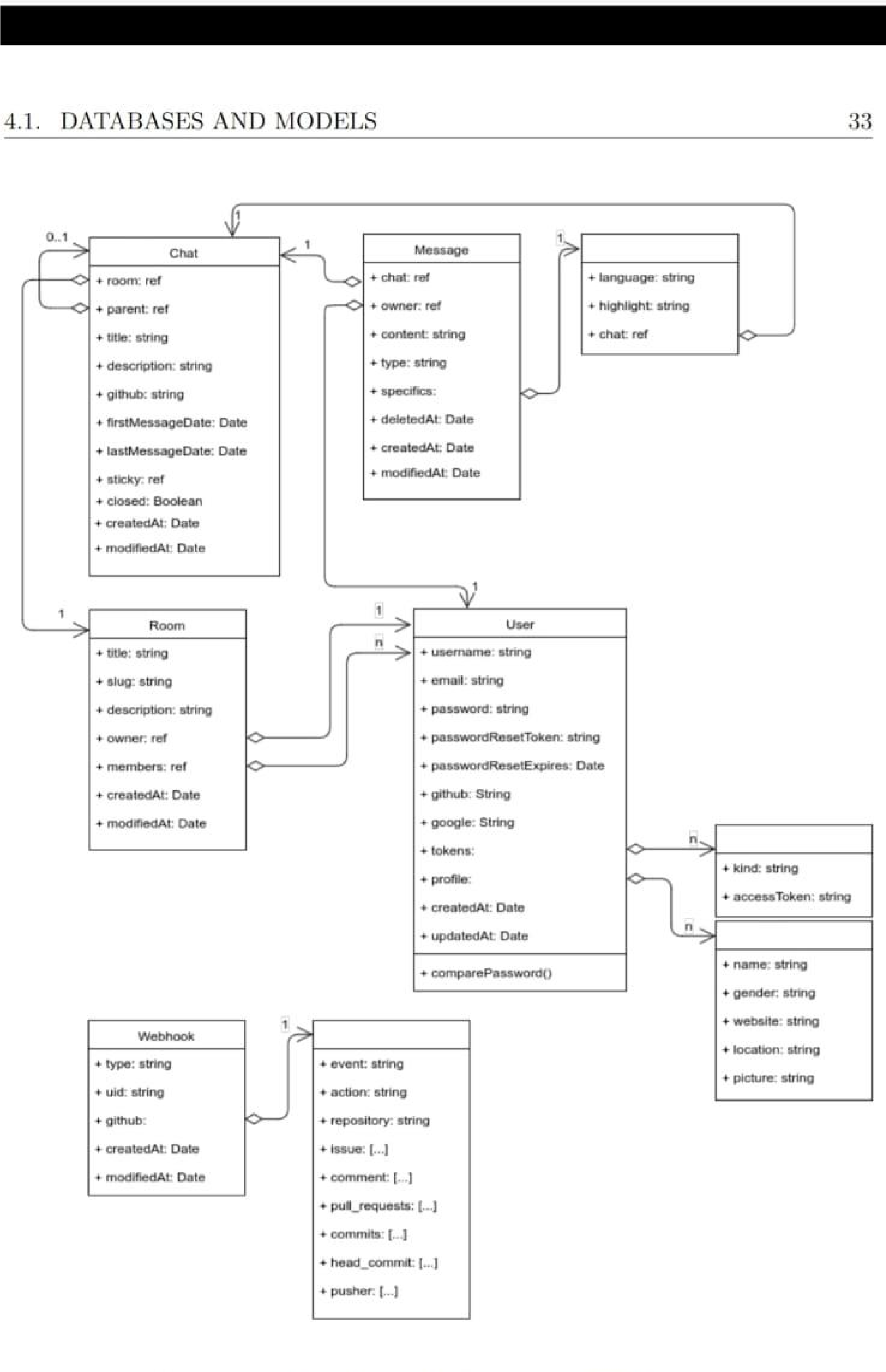
Schema fields:

* \_id: identifier.
* username: friendly identifier.
* email: email address.
* password: encrypted password.
* passwordResetToken: token to reset their password.
* passwordResetExpires: expiration date of the password reset token.
* github: GitHub’s profile id.
* google: Google’s profile id.
* tokens: list of linked services tokens.
* kind: service name (i.e. github)
* accessToken: access token given by the service.

• profile: personal details – name: full name.

* gender.
* location.

14



* website: personal web or blog URL.
* picture: avatar URL.
* updatedAt: last updated.
* createdAt: creation date.

Users collection is indexed by \_id, username, email, github, and google fields. 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 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.

###### 4.1.2 Rooms

Given that we were not going to store rooms as nested data inside the users’ collection, mainly because we were looking forward to referring to them directly, we created an independent collection for them.

In addition, we were expecting many rooms, probably even more than users. Hence it was not a not a good idea to nest them under any other document.

Schema fields:

* \_id: identifier.
* title.
* slug: room URL identifier.
* description.
* owner: \_id of the owner user.
* isPrivate: whether the room is private or public.
* members: array of user \_id who are members of that room.

16

* updatedAt: modification date.
* createdAt: creation date.

Notice that once again we are not storing any of the chats inside it, not even the reference. Although some would argue that it would not be a bad idea, in this case, we preferred storing them on an individual collection given that we were expecting many due to the ability to fork chats.

Other chat applications which set a limit of 10-20 chats per room, should consider either embedding the whole chat object inside their room or at least store a reference to them.

On the other hand, we are storing a reference to the members of a room. That is because we are not expecting more than few hundred users per room and they are also not a clear entity by themselves and the disk space these references take does not look like to be a problem.

When designing MongoDB collections, we always have to keep in mind that the maximum size per document is 4MB (16MB in the latest versions)[10].

The other field which we are also storing by reference is the owner of the room. The reason why we are not embedding the user, in this case, is not because of the size, but rather because the user profile data might frequently be updated which would mean having to update all rooms he owns, apart from the corresponding User.

We are indexing Rooms by \_id, slug, owner, and members.

At first, we thought \_id and slug would suffice since they cover the most common searches: users referring to a chat through its identifier or entering through a direct URL (in which case the lookup would be done by the URL slug).

However, later we realized that users might often want to look up chats which they either own or are members of, which is the reason why we created two additional indexes to cover the owner and members.

##### 4.1.3 Chats

As we stated earlier, our chats were going to be in individual collections. There

17 might be rooms in which their members have few chats, but others might have hundreds (even if that leads to having a few inactive ones).

Once again, we had to think whether it was worth embedding or referring messages inside the Chats collection or keeping them isolated in another one.

In this case, it was evident. We were expecting thousands of messages in any Chat, which would rapidly go over the 16MB that any MongoDB document can hold, even if only storing references. Thus, messages had to be saved in a different collection.

Schema fields:

* \_id: identifier.
* room: identifier of the room it belongs to.
* title.
* description.
* github: GitHub repository name, taken into consideration when creating GitHub specific chats.
* firstMessageAt: date of the first message sent. It is used to determine whether the user has already retrieved all messages of a chat.
* lastMessageAt: date of the last message sent.
* updatedAt: modification date.
* createdAt: creation date.

We are indexing Chats by \_id, and room. \_id is used everytime someone wants to enter a specific chat, whereas the room one makes it quicker to search the chats inside a Room.

18

**REFRENCE**

**1.)**

**https://developer.android.com/docs**

**2.)**

**https://firebase.google.com/docs/**

**3.)**

**https://github.com/codepath/android\_guides/wiki/Consuming-APIs-with-Retrofit**

**4.)**

**https://www.simplifiedcoding.net/**

**5.)**

**https://stackoverflow.com/**

**6.)**

**https://www.slideshare.net/**

**7.)**

**https://code.tutsplus.com/tutorials/how-to-create-an-android-chat-app-using-firebase-cms27397**