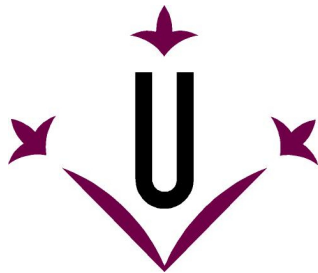# Implementation of a chat application for developers

Gerard Rovira Sánchez

Polytechnic School

University of Lleida

A Final Degree Project presented for the degree of

Computer Engineering

Advisor: Juan Manuel Gimeno Illa

September 7, 2017

Abstract

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 conversation 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 WebSockets, SPA libraries (React), NoSQL databases (MongoDB and Redis), REST JSON APIs with Node, Docker, and more.

Acknowledgements

Contents

List of Figures

Chapter 1

Introduction

1.1 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

- Room roles

- Files sending

- Contacts sharing

- Emojis & animated emoticons

- Audio notes

- Voice calls

- Video calls

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

Technical

- MarkDown and limited HTML formatting

- Sticky/pinned messages

- 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 CodePen and JsFiddle results

GitHub

- Code snippets sharing (i.e. by pointing out initial and ending line of any repository file)

- Specific repository chats

    - Watch commits

– Watch and discuss issues and pull requests

– Sync issue discussion with GitHub

1.2 Document Structure

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.

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`

- 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.

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 CodePen or JSFiddle.

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.

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

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.

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 be 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.

Chapter 3

Phase 2: Planning, Prototypes, and Technology

3.1 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.

3.1.1 Scrum

Scrum is one of the most popular Agile software development frameworks. It is iterative and incremental.

Scrum's objective is to create working versions of the product in a short amount of time. The final product will get more and more complete in every iteration. One of Scrum's main features is that it takes for granted that requirements can vary at any given point and that few requirement changes should not be a hassle to have the product completed within a limited time frame.

When working with Scrum, those who are involved in the project start off with a product backlog, which is a sorted list of jobs by priority. We presented our initial list in the next section, where we wrote about use cases and scenarios.

Scrum works with sprints, short periods of time in which certain amount of work is meant to be completed. After each sprint, team members have to come together to evaluate the work that has been done during that time and the work that is left to do, as well as decide what has to be done in the following sprint.

We defined our sprints to be as long as two weeks. We discussed our project's progress with our tutor at the end of every sprint.

Given that a final degree project is meant to last a semester, we were going to have about nine sprints.

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. 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:

- 1-2 days: 1

- 3-4 days: 2

- 5-6 days: 3

- 7-9 days: 4

- 10+ days: 5

To sort both the product backlog and sprint backlog lists, we relied on a third number, the priority, which was simply the result of the value minus the time cost.

Nonetheless, in some cases, we had to make exceptions due to user stories dependencies. For example, sign in and sign up features had to be implemented the first, since we needed user information to properly identify the room owner or the message sender.

Our initial product backlog list was the following:

#	User story	Value	Time	Priority
1	As a user I want to be able to sign up with a unique username (that serves as an identifier throughout the platform).	1	4	10*
2	As a user I want to be able to sign back in.	1	4	10*
3	As a user I want to be able to log out when I will no longer use the platform on a given device.	1	1	10*
4	As a project manager I want to be able to create a new room for a project of mine.	5	2	10*
5	As a user I want to be able to join rooms, previously created by the project manager.	7	1	10*
6	As a user I want to be able to create discussion chats.	6	2	10*
7	As a user I want to be able to enter the chats in the room I'm in.	6	1	10*
8	As a user I want to be able to read real time text messages in a chat.	10	4	8
9	As a user I want to be able to write text messages, which will be displayed to other room members in real time.	10	2	8
10	As a user I want to be able to format messages (bold, italics, links, ...).	8	2	7
11	As a user I want to be able to share images when chatting.	8	2	7
12	As a user I want to be able to share code snippets.	9	3	7

13	As a user I want to be able to stick a chat message on top of the chat.	7	2	6
14	As a user I want to be able to edit a chat.	7	2	6
15	As a user I want to be able to highlight some parts of a snippet.	7	2	6
16	As a user I want to be able to link a chat with a GitHub repository, so that chat members can watch its updates in real-time (pull requests, issues, commits, comments, ...).	9	6	6
17	As a user I want to be able to fork a chat based on an activity message.	8	4	6
18	As a user I want chat information updates to be displayed in real time.	6	1	5
19	As a user I want to be able to merge a fork after the forked chat discussion has ended.	7	4	5
20	As a user I want to be able to read URLs' description to know what the page is about before entering.	6	2	5
21	As a user I want to be able to post emojis on a chat.	6	1	5
22	As a user I want to be able to send files of any kind.	7	4	5
23	As a project manager I want to be able to kick members of a room.	6	2	5
24	As a user I want to be able to post real time polls on a chat.	6	4	4
25	As a developer I want to have access to a public API, so as to be able to build an application based on this one (such as bots integration).	9	30	4
26	As a project manager I want to be able to remove a room.	4	2	3
27	As a user I want to receive desktop notification when someone has sent a message, and I was not on the chat page.	3	1	2
28	As a user I want to be able to change my avatar.	2	2	1
29	As a user I want to be able to change my status (online, offline, away, busy, ...).	2	2	1

30	As a user I want to be able to find contacts by connecting to social networks.	2	2	1
31	As a user I want to be able to do group calls.	6	20	1
32	As a user I want to be part of a reputation system (based on my chat and GitHub activity).	6	15	1

* Inflated priority due to dependencies.

Later on, when we were implementing the project, we found out a few requirements that were missing and we noted them down.

Again, that was not a problem and it was in fact expected. After all, that is the main reason why are were using Agile development.

The new requirements table looks like the following:

User story	Value	Time	Priority
As a user, it would be handy to have a landing page that points to existing chat rooms.	8	2	7
As a user, I want to have MarkDown support for messages, which includes at least bold, italics, tables, lists, and hyperlinks.	8	2	7
As a user, I want chat messages to auto scroll, other than having to manually scroll them down once I get a new message and the chat container is full.	7	1	6
As a user, I want activity messages to auto scroll, other than having to scroll them down manually once I get new updates and the messages container is full.	7	1	6
As a user, I want the chat text input box to automatically resize itself if the text message is too large and doesn't fit the default one.	7	1	6
As a user, I want to be automatically reconnected to the chat if my connection drops.	5	3	3
As a user, I would rather not have all conversations loaded at once, but rather have old ones retrieved whenever I need them.	4	3	2

As you might have noticed already, product backlog, and consequently sprint back-
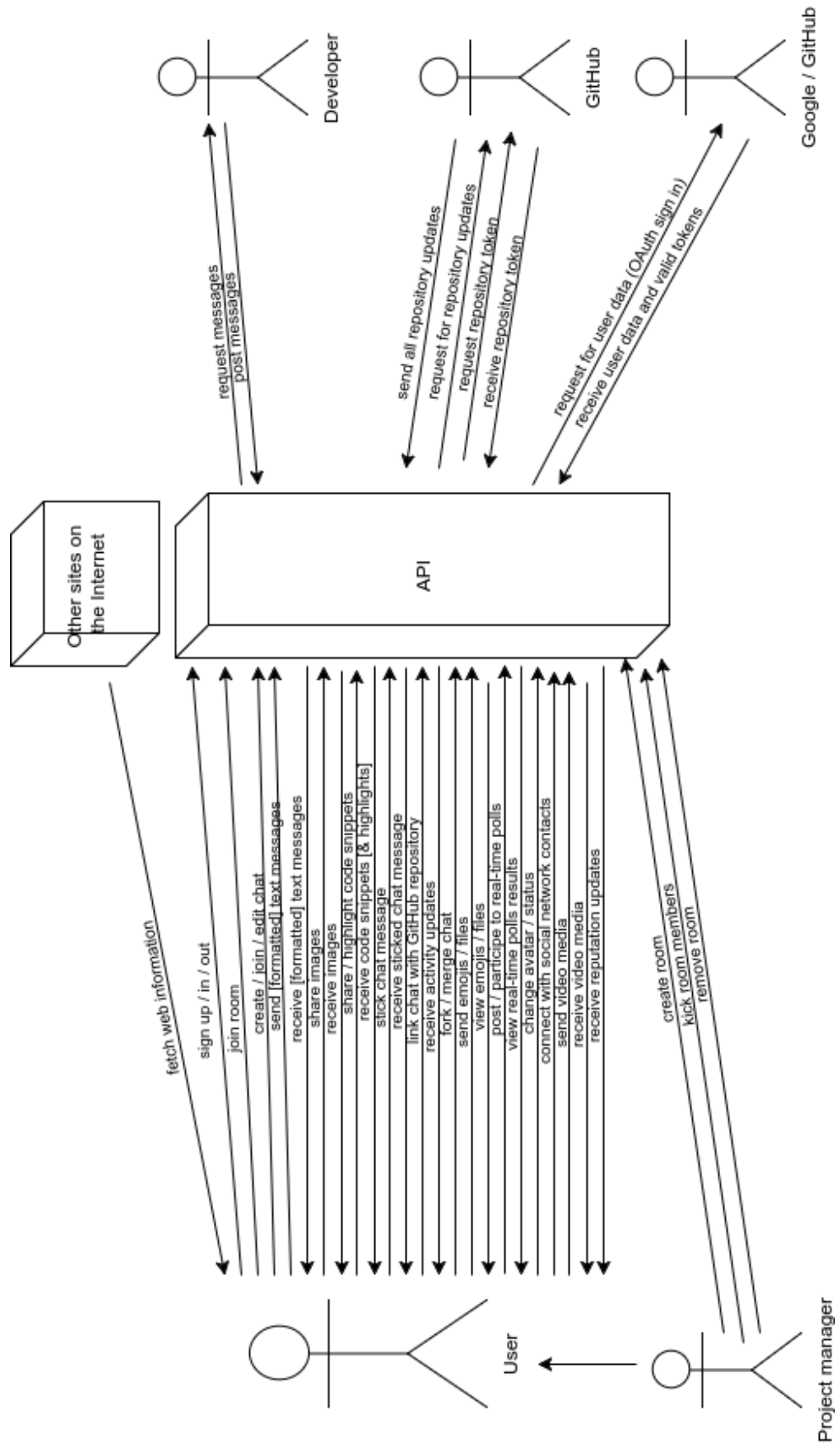
Figure 3.1: Initial context diagram.

logs too, go hand to hand with the features described in our Ideal platform. It makes sense that we have plenty of user stories since they describe the characteristics that would be desirable to have, but that also means that a few of them will remain undone.

We were able to complete all user stories which had a priority of five and above.

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
0-4	Initial report with app objective, time schedules, and other valuable starting data. Build application boilerplate which will serve as the base of our app.
5	User registration and authentication.
6	Implementation of public rooms, which users will be able to join freely.
7	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.
8	Desktop and site notifications, user status (online, busy, away), ability to type and send emojis, MD and limited HTML formatting.
9	Implementation of room roles.
10	Ability to stick messages, highlight code, improve previously sent code, and proper code formatting depending on the language it has been typed in.

11	Documentation (handling mid-term report the following week). Review/finish parts of current logic and design that might have been postponed.
11	Dedicated GitHub repository chat: watch commits, issues, and pull requests.
12	Discussions forking. Users will be able to fork a chat based on a previous comment/item, issue, commit, pull request.
13	Reputation system based on users activity (on the platform and on GitHub), such as pushing commits or creating new issues.
14	Private rooms, including administrator(s) invitations to join a room.
15	Deployment of the API, database, and client on a cloud server.
16	Documentation.
17	Documentation.

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).

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).

Sinon: 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 WebSockets, 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[6]: you can access nested data without having to perform any join.

- They are faster[6]: nested data is stored in the same place and can be consulted without any additional query.

- They scale better[7] 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[8].

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 schemaless 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 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.

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 (known as composition[9]).

```
function RootComponent(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 transpiled into HTML components.

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

is transpiled to

```
<input class="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 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.

Parent component

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

TextInput component

```
function textInput(props) {
  return (
    <fieldset>
      <label for="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

```
function inputChanged(event) { ... }


function ParentComponent(props) {
  return (
```

```jsx
    <form>
      <TextInput name="email" onChange={inputChanged} />
      <TextInput name="address" onChange={inputChanged} />
      ...
    </form>
  );
}
```

TextInput component

```jsx
function textInput(props) {
  return (
    <fieldset>
      <label for="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 V(iew) 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. React'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 transpiles 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.

Enzyme: a library which eases React components testing. This library was made specifically for React and enables us to simulate components like if they were rendered on the DOM. It is often used to test button handlers or elements which should appear only under certain conditions.

Sinon: A JavaScript testing library. Since it is not Node.js specific, we are also using it on our front end side to make testing easier.

Express: Express on the client side? Yes!, to provide server-side rendering.

SPA applications have no content in the HTML file that is sent to the client. All the content (and logic) resides inside the one or many JavaScript chunks that the site may have, which means that browsers have to download these file(s) and execute them prior being able to display anything useful into the user's screen. That can take up to a few seconds depending on the sizes of the main JavaScript chunks.

By rendering the first requested page on the server, we can answer back with the preloaded HTML (and even CSS), and even though the navigation will be very limited while the chunks are still downloading, they will be able to read that page's content just as if the page had been fully loaded.

Apart from that, some search engines are limited to reading HTML. Since all our content resides inside JavaScript files, these will see no content. Consequently, they will not take into consideration any of the content headings or other types of keywords when indexing it. Fortunately, that is not the case for Google anymore.

A combination of ReactDOMServer and Node.js enables us to return a rendered view of any page on our site, along with the JavaScript chunks endpoints.

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.

3.4.4   Continuous Integration

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.

We are using few of these:

Travis and CircleCI: The two most important ones. They are both Continous Integration systems that make sure our application builds successfully and all its tests pass.

CircleCI was added by the end of the project, and it is also in charge of packaging our source code into a Docker container and deploying it into a cloud-based service. We described this in detail in the deployment section of the implementation chapter.

David-DM: Watches our NPM project dependencies, and warns us whenever a dependency is obsolete or deprecated. An obsolete or deprecated dependency might have severe issues which were corrected later in time, and so it is recommended to always stay up to date.

Codecov: Reports the current tests coverage state and gives a total coverage per-

centage. It offers a nice web interface to watch how the total percentage is changing over time, as well as individual percentages for each of the project files.

CodeClimate: Analyses the project files, and gives a rating 1-4 rating depending on its quality. For example, your total mark gets decreased if any of your source code files were too low, or you have unused variables or functions.

By using their web interface, you can check each of these quality mistakes and correct them to make your code cleaner and more readable.

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 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
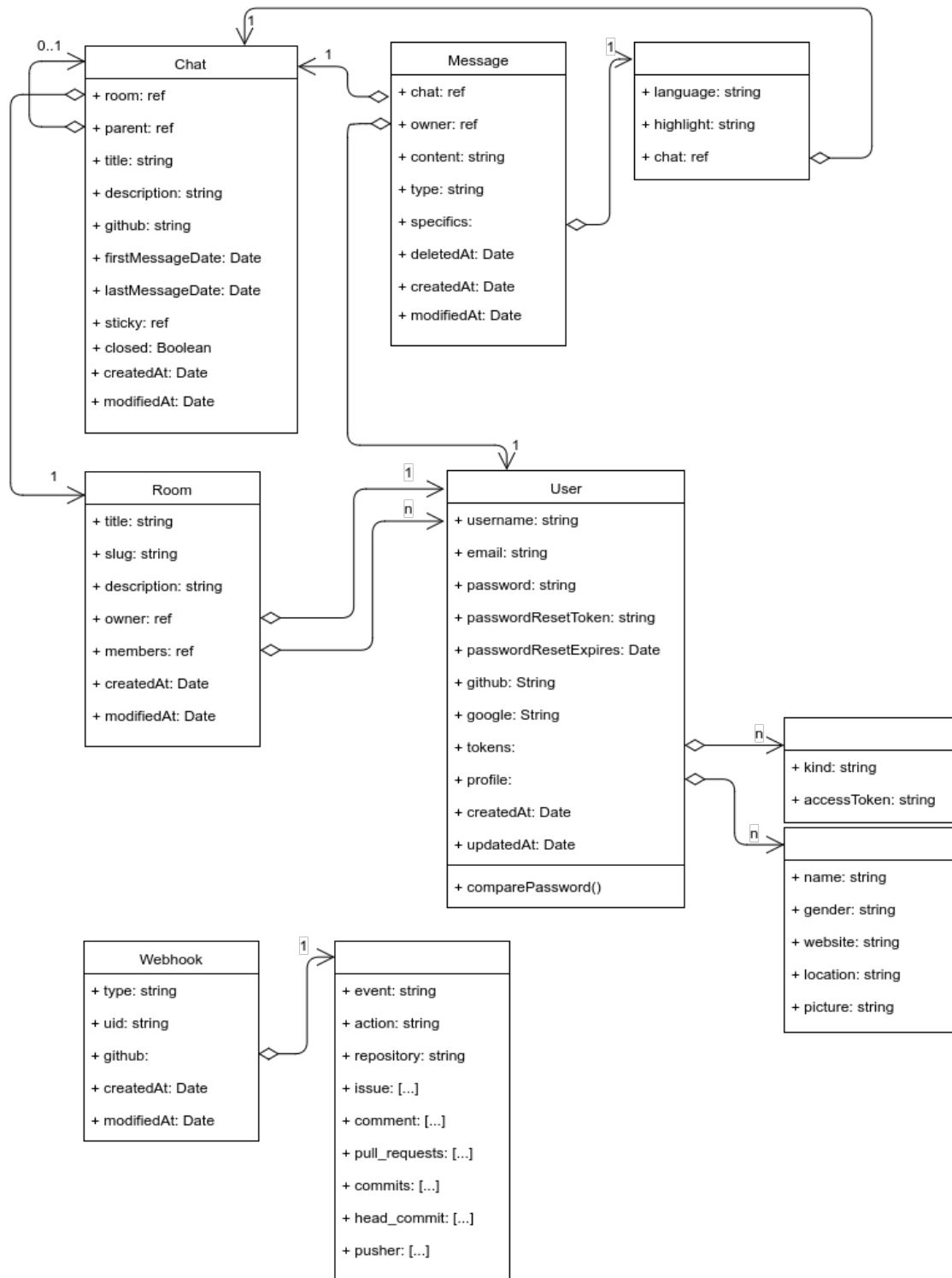
Figure 4.1: Final MongoDB database UML diagram.

- name: full name.

- gender.

- location.

- – 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.

- 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 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.

4.1.4   Messages

We were expecting thousands of messages per month, so the right way to store them, according to the MongoDB official documentation, is in an individual collection. In a production environment on which we were expecting even millions of them, we might have to consider sharding the data to avoid bottlenecks, which is a topic which we have briefly covered in the Evaluation chapter.

This case is similar to the Rooms or Chats ones, but this time it is taken to the extreme, "One-to-Squillions"[11]. We were no longer just expecting to store thousands in the long-term run, but we were expecting to store thousands at a fast growing pace.

We can summarize our Messages needs as follow:

1. Ability to store hundreds of messages per hour.

2. Ability to retrieve thousands of messages per hour.

3. Ability to retrieve messages in chronological order (most recent first).

Notice that we are expecting to read more than to save. That is because a few chat peers are likely to retrieve recent messages more than once, and while a message is only stored once, several members are likely to read it numerous times. Thus, we wanted to design a collection schema which favored reads over writes.

Moreover, we would never want to retrieve all messages at once. Not only it would be impossible for the user to read them all, but also we would not be able to handle the load if we did that for Chats having many messages.

Schema fields:

- _id: identifier.

- chat: identifier of the chat it belongs to.

- owner: sender (User) identifier.

- content: text.

- type: content type.

  - language: code language, if the message type is code.

  - highlight: lines to highlight, if the message type is code.

  - chat: reference to the parent Chat, if the message is a fork.

- deletedAt: deletion date. Content will be removed on deletion, but their peers will be aware that the User sent a message at that time.

- updatedAt: modification date.

- createdAt: creation date.

As simple as it seems, this structure has been proved to work out for up to 1,000,000 concurrent connections[12].

We have indexed messages by _id and chat + createdAt. The first one helps when looking for a specific message, whereas the second composed index works out well when looking for past messages. We have composed the date with the chat to filter only the messages which belong to a particular chat since we will never be interested in mixed chat messages.

4.1.5   Webhooks

During the implementation of the application, we got to the point when we needed to store data from GitHub WebHooks, which we will go through into why we needed them in section 4.2.9.

GitHub WebHooks only send the information once per repository, so we had to make sure that the data was stored in a way that any chat linked to that GitHub repository could access GitHub updates.

Also, the information had to be able to be fetched and stored quickly, since GitHub updates tend to be numerous and we wanted to notify the Chat users in real time.

This case resembles the Messages one, though this time we would be dealing with messages sent by bots.

Schema fields:

- type: webhook type, in case there was more than one provider.

- uid: unique identifier of the webhook message (sent by the saving/updating).

- github: GitHub specific fields, such as repository name or action.

- updatedAt: modification date.

- createdAt: creation date.

We have indexed webhooks by _id and _id + github.repository, which is similar to what we did with messages. Nonetheless, we would probably have to create more indexes if we had more providers, since _id + github.repository only suits for GitHub ones.

4.2   Features

In this section, we will present the implementation process of the most relevant features, which are part of the user stories we described earlier.

Each of the subsections goes over a particular characteristic, detailing the relevant parts of both the back end and front end development.

We have sorted them in chronological order: we have started describing the process of setting up the development tools, especially Express and React, and we have ended up with the GitHub activity implementation.

4.2.1 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[13].

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 URLs lack of some CRUD[14] 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-urlencoded into JavaScript objects, executing certain controllers when a route is matched and returning a value).

Express, as a fully-fledged framework it is, is also able to a start web server by itself (with all its logic in it), which was OK at first. But later on, we moved that responsibility to a third party library (http) which allowed us to run both the AJAX API and WebSockets on the same port.

We are using Socket.io to work with WebSockets. We will go through it in the messaging subsection. When it comes to the configuration it has access to session data as well as run with the http library (already working with Express).

Finally, the http package starts both Express and Socket.io on a single port, which is advantageous when preventing CORS[1] restrictions either when developing or in production.

Having decided the technologies we were going to use, how we were going to communicate with the client and had our back end core prepared, we were ready to move onto our server folders structure. We wanted something scalable which was easy to maintain since there was a lot to be written into that application and we wanted to prevent having to restructure everything in the middle of the project.

Our server folder structure was looking like this at the very early stages:

```
.
|-- bin
|-- config
|-- data
|-- data-session
|-- docs
|-- node_modules
'-- src
    |-- controllers
    |-- helpers
    |-- middleware
    |-- models
    |-- routes
    '-- sockets
```

It eventually grew somewhat more, but fortunately, the changes ended up being minimal.

bin: batch files or other executables.
config: user configuration files, that are not bound to change, such as database

---

[1]CORS: Cross-Origin Resource Sharing https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS

settings or application keys.

data and data-session: where the MongoDB and Redis will store their data.

`node_modules`: npm modules.

`src`: contains all application logic.

`src/controllers`: request handlers.

`src/helpers`: utilities that can be used anywhere in the app, which will generally be used more than once.

`src/middleware`: node.js express framework can use middleware to handle requests, before they finally reach a controller.

`src/models`: databases models which we went through in the previous subsection.

`src/sockets`: socket connection handler (including event senders and listeners).

You might have noticed that there are no test folders. We have written tests, but we thought that keeping them next to the utility was more advantageous than writing them on a separate folder: you avoid rewriting the entire folder tree, you can figure out instantly the source code files that are missing tests, and if you ever need to move/delete that file you rapidly notice that the test file does not match any of the source code files in that folder.

```
'-- moduleFolder
    |-- index.js
    '-- index.test.js
```

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[15]. 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 can 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.

There is a catch when using PATCH though, even though it works well for adding and updating fields, it is impossible to state that you want a field removed through an object[16]. That is why RFC 5741[17] standard has such complicated format.

Nonetheless, we felt that PATHching was too good to have it replaced for PUT, so we ended up supporting an alternative extremely simple version of PATCH: A PATCH request which treats null values as removals, which is also what Google Cloud API does[18]. Although this option is quite popular, it will most likely never make it to the standard since we are not distinguishing between a null value and a removal, but it will work just fine for us.

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 WebSockets), 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.

This is how our folder structure looked at the very early stages:

```
.
|-- dist
|-- src
    |-- components
    |-- containers
    |-- helpers
```

```
    |-- redux
    |-- redux
        |-- middleware
        '-- modules
    |-- routes
    |-- styles
    '-- tests
'-- webpack
```

`dist`: the files that are meant to be distributed, mostly generated by webpack module bundler. It contains an index.html, that will be the start point for anyone who accesses our site and the JavaScript bundles containing the application logic (including required external libraries).

`src`: contains all application logic.

`src/components`: "dumb" components[19] which do not interact with Redux.

`src/containers`: contains connected components, components that interact with our Redux storage.

`src/helpers`: general JavaScript code that does not fit anywhere else (a sort of utilities). Currently storing our own implementations of Promise based AJAX Fetching and WebSockets Client.

`src/redux`: everything about Redux.

`src/redux/middleware`: Redux enhancements, like supporting the special object action to fetch straight from the API.

`src/redux/modules`: Redux reducers, one for every distinct topic on the site.

`src/routes`: application routes linked to their appropriate components.

`src/styles`: contains global application styles and SCSS variables.

`src/tests`: Global tests configuration.

`webpack`: Webpack configuration files, containing at least 2 configuration files: the development and the production one.

The development one generates hot-reloaded bundle.js[2] files, barely optimized but that build very quickly, and the production one which generates the lightest bundles to fit into a production environment.

Individual tests, just like we did with the back end, are located next to the implementation. For example, a component.js has a component.test.js on the same folder.

During the development, we also noticed that as the application was getting bigger

---

[2]Webpack React Hot-reload: changes done on the source code while the application is running are pushed to the browser through a web socket as small JavaScript update packets, which prevents having to re-reload the page to see changes and keeps current React state.

so was the size of the bundle, and we urged to find a way to lower its size. That was when we decided to split the main bundle file into specific bundles that are only retrieved when needed. avoid we chose the split point to be the different routes.

As a result, the router ended up not only the one deciding which component to load but also fetching the required files before switching to another page.

4.2.2   Authentication

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

Server

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).

Next, we had to decide how the client was going to inform the server of the signed in user. Most API services include the authentication token in query parameters, but on the other hand, most web applications tend to use cookies for this, and they send these cookies on each request.

We went for the second one, mainly because we thought it would be easier for our front end to handle.

Our client application would also need a specific endpoint where to verify whether its current stored token was valid and to get to know the user that was behind it in order to display the current logged in user and to enable certain features which are only available to users.

We used `users/whoami` for that.

When it comes to the authentication logic, we used Passport for the most part. Passport hides all the complexity behind OAuth and OAuth2 into a generic API that works the same way for hundreds of providers.

We have also used it to handle our local authentication, even though it was not strictly necessary, just for the sake of consistency.

Apart from that, even though the Passport library did the hard work, we still had to control what happened after a successful callback: whether an account had to be linked with another, create a new user, or prevent the user from signing in because it had not passed our validation checks.

We consider account linking to be quite important to enhance the user experience on our site, given that we use the provider API key to gather extra information over the time. For this reason, we try to link the user account with as many providers as possible if that this is a possibility. At the moment a user who has created his account with the web form can end up with both Google and GitHub accounts linked to themselves.

One thing that might be worth pointing out about the previous diagram is the fact that *"Signed in user is linked to this provider ID"* is before *"OAuth ID found"*. Logically, it would make sense that they were the other way around, but due Passport middleware always checking the user first at every request, we can avoid up to an extra query to the database.

There is not much to say about the data validation, other than we are not being very strict. Local email has to be a valid email address and the password has to have a minimum of 4 characters.

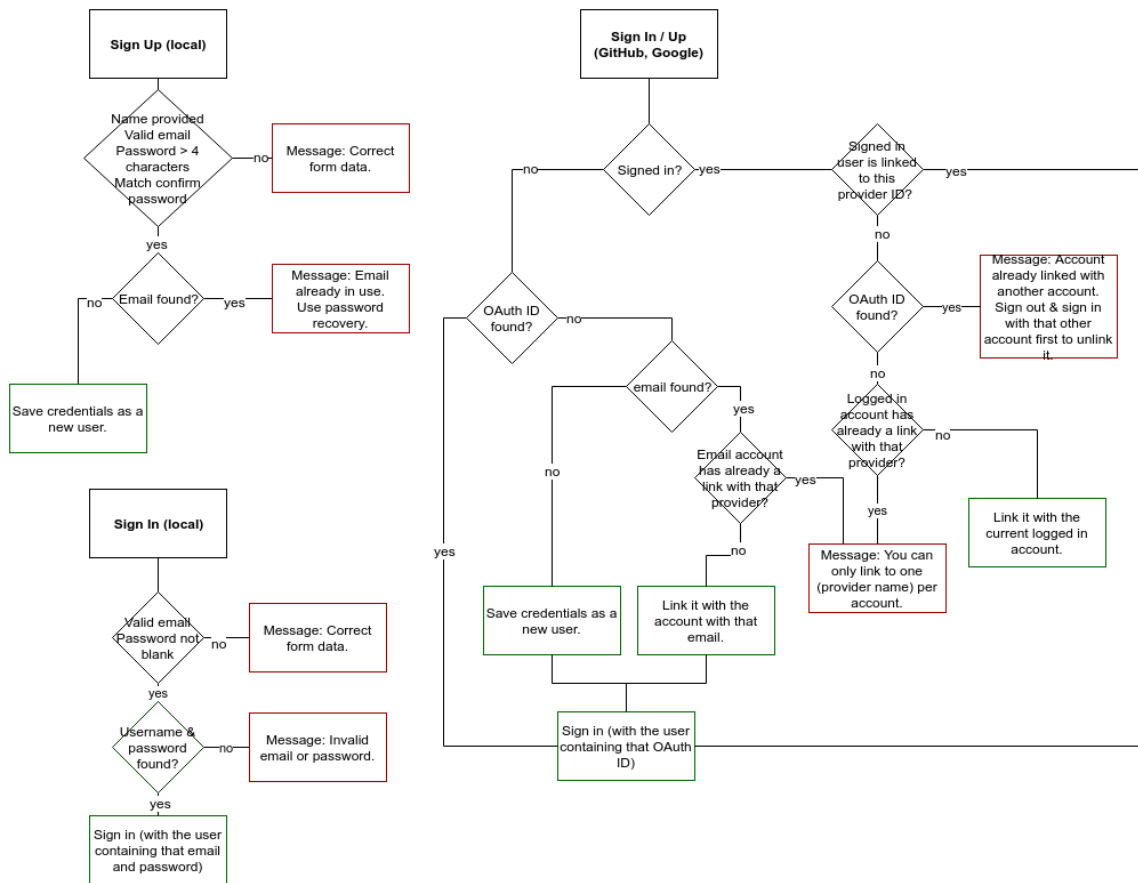Finally, there is the sign out page (`/auth/signout`), which does nothing but remove
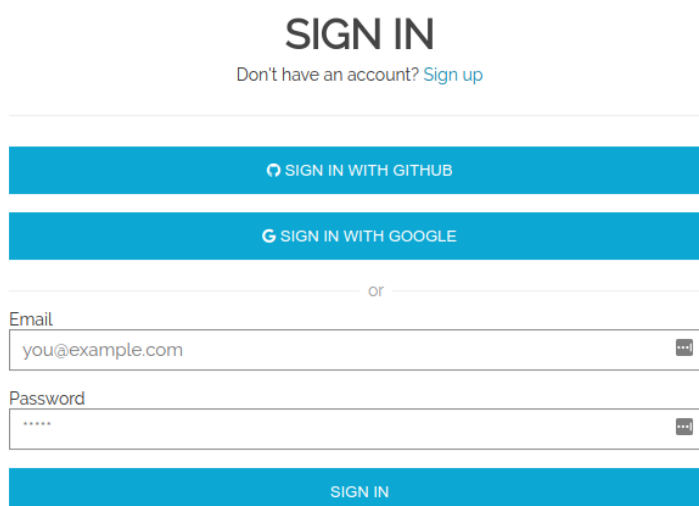
Figure 4.2: Authentication flow diagram

the link between the session and the user.

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.
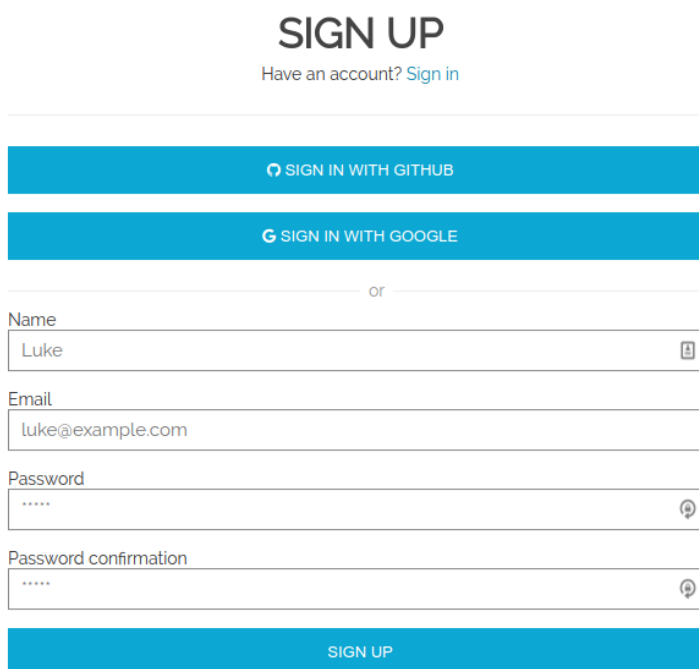
Similarly, there is a button with no fields associated that starts the sign-out process.

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.

Figure 4.3: Sign in form

Figure 4.4: Sign up form

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.

Our local authentication, which works with a user-provided email and password, works with a form and state is updated just after checking the credentials with the
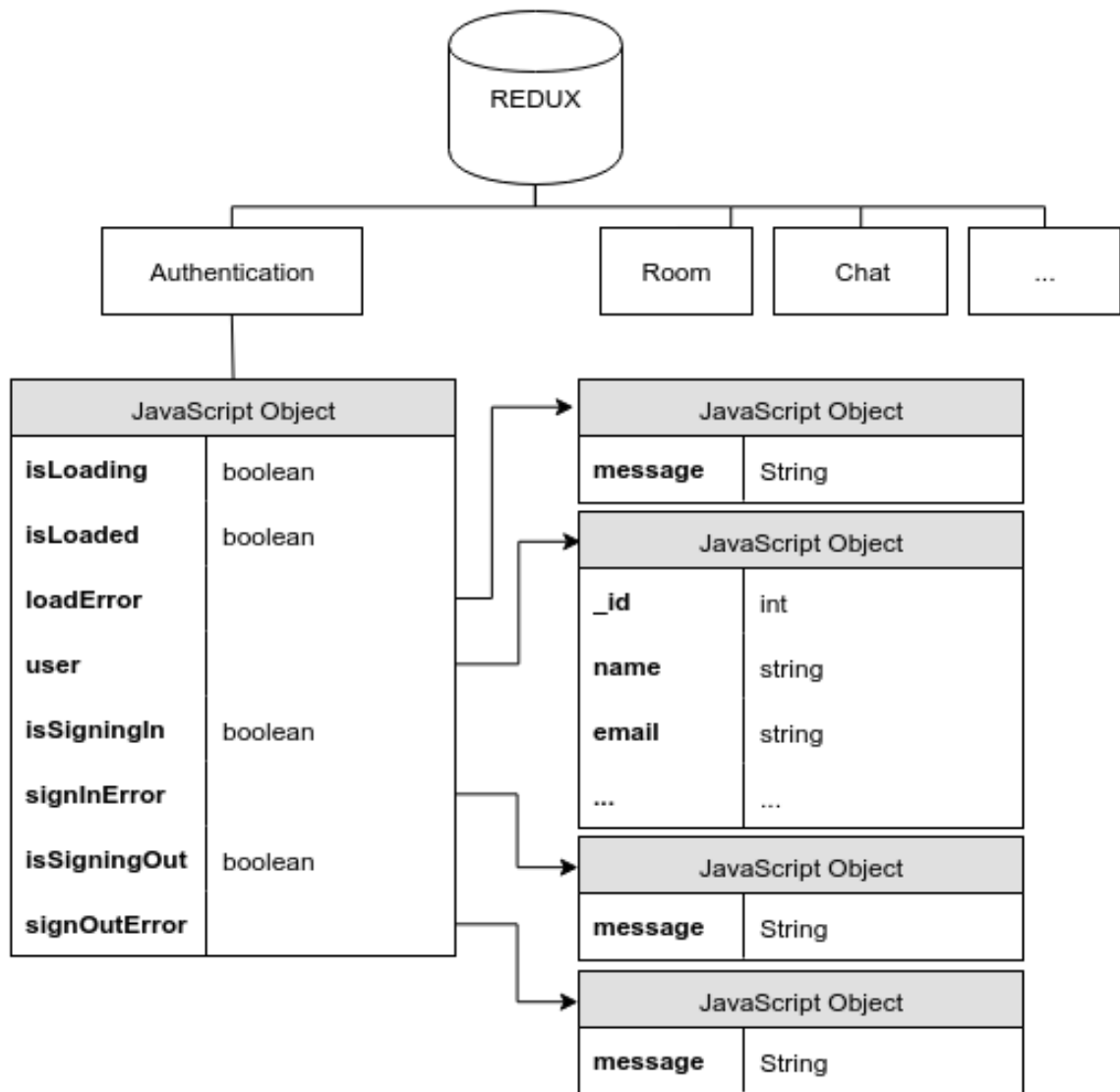
Figure 4.5: Diagram of Redux authentication module

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.

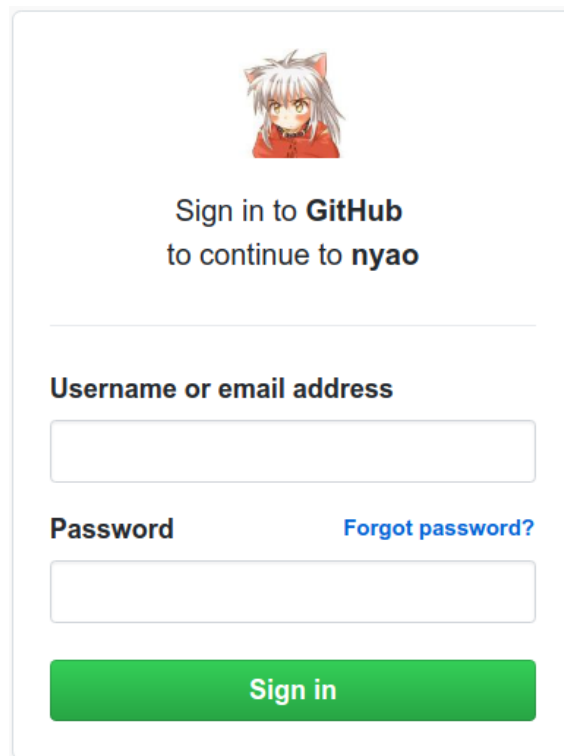Figure 4.6: GitHub OAuth authentication.

4.2.3   Rooms, Chats & Messages CRUD

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 methods as explained in section 4.2.1.

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[20].

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.
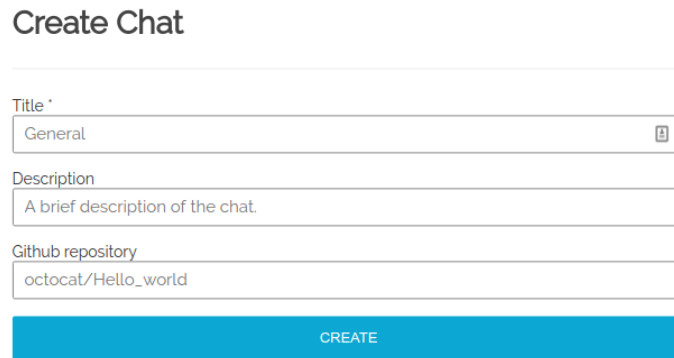
Messages

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.

The client consumes these API endpoints, and requests or displays the appropriate data to create or read the different site entities, by using forms or tables of data (as displayed in figures 4.7, 4.8 and 4.9).

Figure 4.7: Create room form.

Figure 4.8: Explore current platform rooms grid.

Figure 4.9: Create chat form.

Although we created pretty much AJAX endpoints for everything, we left messaging ones out for now. In the following section, we will describe how we are going to use WebSockets for this instead of the AJAX ones, to make sure the messages are sent and delivered in real time.

Other than the typical CRUD, there is one relevant thing worth noting about the client implementation. AJAX requests are always executed from the same method (fetch) with a certain configuration, that includes cookies in the request (which by default they don't). Since that is unlikely to change, we abstracted it into a Redux middleware that all it requires is a URL and an optional body content to work. In the messaging section, we'll see the real benefits of doing so when implementing a similar for WebSockets.

4.2.4   Messaging

All our messaging dependencies were complete. It was time to start the messaging system.

We will use WebSockets for this. When exchanging real-time messages, especially when receiving, AJAX is simply not adequate for the job. Not only is slow but also it is not event driven, you never know when a new message has arrived unless you send a request to the server for new updates.

WebSockets make it possible to maintain a real time, low-latency and bi-directional connection between the client and server. The persistent bi-directional connection is what makes it possible to receive server messages at any time. It also avoids having to send redundant HTTP headers with every request, which can make a noticeable difference in the application's bandwidth usage[21].

Server

On the server side, Socket.io, the high-level WebSockets library we are using, is always listening for new connections. It offers a limited functionality over what AJAX does and it is mostly meant to be used to send/receive real-time updates.

When using WebSockets, we take for granted that the user will already have an account on the site and they will already be signed in. Otherwise, the socket connection is rejected.

For messaging, we declared the following events:

Event	Data	Description
disconnect		Closes socket connection.
EnterRoom	Room slug	Petition to enter a room, which will subscribe the user to new incoming messages.
SendMessage	Object containing chatId and content	Sends new message to the given chat.
ReceiveMessage	Message object	Whenever a new message is sent in a room chat, the server will emit this message to all members in that room.

There's one significant thing to note here, and it is that contrary to AJAX, WebSockets do not expect a response in return whenever they send in data. By using raw WebSockets this would be a big disadvantage, the client would never know whether any of its requests failed due to server-side validation. For example, when an empty message body has been sent. Thankfully, Socket.io provides something called "acknowledgment", which is a sort of AJAX-like responses on which the server can return any kind of data to events.

For events on which the client is just supposed to listen to new information, such as ReceiveMessage, no acknowledgment is possible, but we also expect no errors. The client has to have previously requested to receive these sort of information, and it will then receive any errors that might prevent it from receiving that content. Otherwise, no information will ever be sent from that event.

Just as we mentioned in the chapter above, the DRY principle[20] helped us get rid of duplicated logic between our WebSockets and AJAX. The duplication came from WebSockets events that did the same as similar AJAX routes, for example sending messages. In this case, we abstracted the controllers[22].

Client

We are also going to use Socket.io for this part, in this case, its client implementation (Socket.io-client) which works hand in hand with it.

The first and foremost problem we faced was storing our Socket.io state. We never had to do this with AJAX because HTTP has no state, so we only had to handle its responses.

In order to find a solution which fitted our case, we had to have to global look at the current component tree where it was meant to work in.
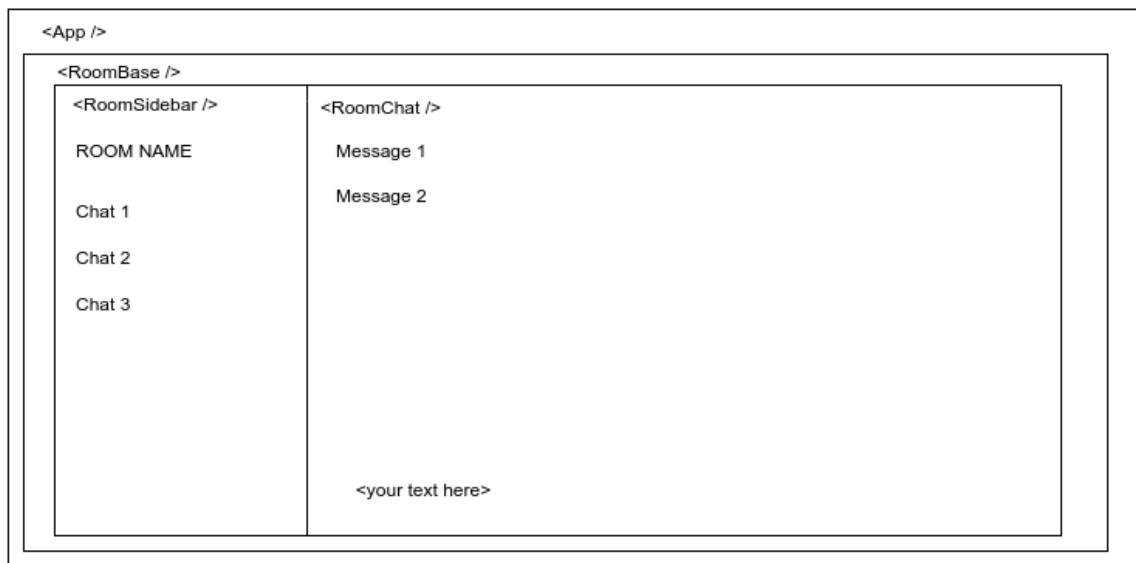
Figure 4.10: Room React components overview (success case).

Considering the components that take part in a chat room page (Figure 4.10 displaying success case), we had a few different possible solutions:

- RoomChatBase container to store the initialized Socket.io state and send it to all its children that have to interact through WebSockets.
  This one is by far the easiest to implement, but it would require all actions to take an extra parameter which would be the Socket.io state. Even worse, we would not have access to the Socket.io state outside the room, so we would have no control on that state whatsoever when that page was left.

- A Redux actions file to store the Socket.io state.
  If we did it this way actions would not have to keep that extra parameter proposed in the previous solution and they would still have access to the Socket.io state. However, we felt that this solution was not the most appropriate one because it would force all different socket actions to be stuck in a single file (when we are currently separating actions by topic to support modularity).

- A specialized Redux middleware that proxies access to a Singleton Socket.io instance. We went for this last one, as explained below.

The middleware approach is similar to what we just did in the previous section with AJAX. The main difference relies on that it required no instances but merely a request configuration since it had no state at all, and Socket.io does require us to keep track of it.

By following the middleware solutions, we get to the cleanest and most reusable solution of them all. Redux actions no longer have to know about the WebSockets library nor the specific configuration we are running it under. All they have to "dispatch" is the content they want to read or write from the server, and the middleware will do all the hard work.

That is how our message Redux action looks like:

```
export function send(chatId, content) {
  const message = { chatId, content };
  return {
    type: 'socket',
    types: [SEND, SEND_SUCCESS, SEND_FAIL],
    promise: (socket) => socket.emit('SendMessage', message),
  }
}
```

As we can see in the snippet above, we parametrize our own socket client, and we send in the desired actions and content by using our own methods. Behind the scenes, these methods will eventually call Socket.io methods on its Singleton instance. That is also known as the Strategy pattern[23].

Since we already have two middleware working with requests, we added an extra type field to distinguish between AJAX and WebSockets petitions. Socket middleware, which in our case is executed before the AJAX one, will skip any action that does not have type equal to 'socket'.

We also used this opportunity to proxy the Socket.io library into a Promise-based one (instead of mirroring their callback methods), which is how we are dealing with most asynchronous content on both the client and the server. It makes it easier to handle both error and success cases.

Having solved that, it was about time to start producing.

Overall, there are several steps which a room has to follow before it can eventually assert that it is fully loaded. Some of them can be performed in parallel, some others

require to be done in a certain order:

- Retrieve user information

- Connect to socket

- Socket to enter room

- Socket to listen for new messages

- Retrieve room information

- Retrieve room chats

- Retrieve chat details (of the active one)

- Retrieve active chat old history

Which leads us to a half sequential half concurrent diagram. Starting with the authentication stuff, handled by the global App component, it is following by the RoomBase component which will start the socket connection. Once the RoomBase has established the socket connection and has subscribed to the room messages, it starts both RoomSidebar and RoomChat components which will load their specific information: room information, chat rooms and chat information and messages history if a chat is active (see Figure 4.11).

An important decision we had to make is whether to start with the socket connection or initially load the content with AJAX and move to WebSockets as soon as the connection had been established. To the end user, this would result in a faster-loaded chat page.

We ended up waiting for the WebSockets connection to start, as the socket connection was not taking a long time to establish and the added complexity made it not worth the time to implement in our application for the time being.

When it comes to displaying the messages, we are accepting limited MarkDown formatting. The transpiling from MarkDown to HTML is performed by the Marked[3] JavaScript library.

The result of this is displayed on figure 4.12.

4.2.5 Room & Chat updates

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

---

[3]Marked `https://github.com/chjj/marked`
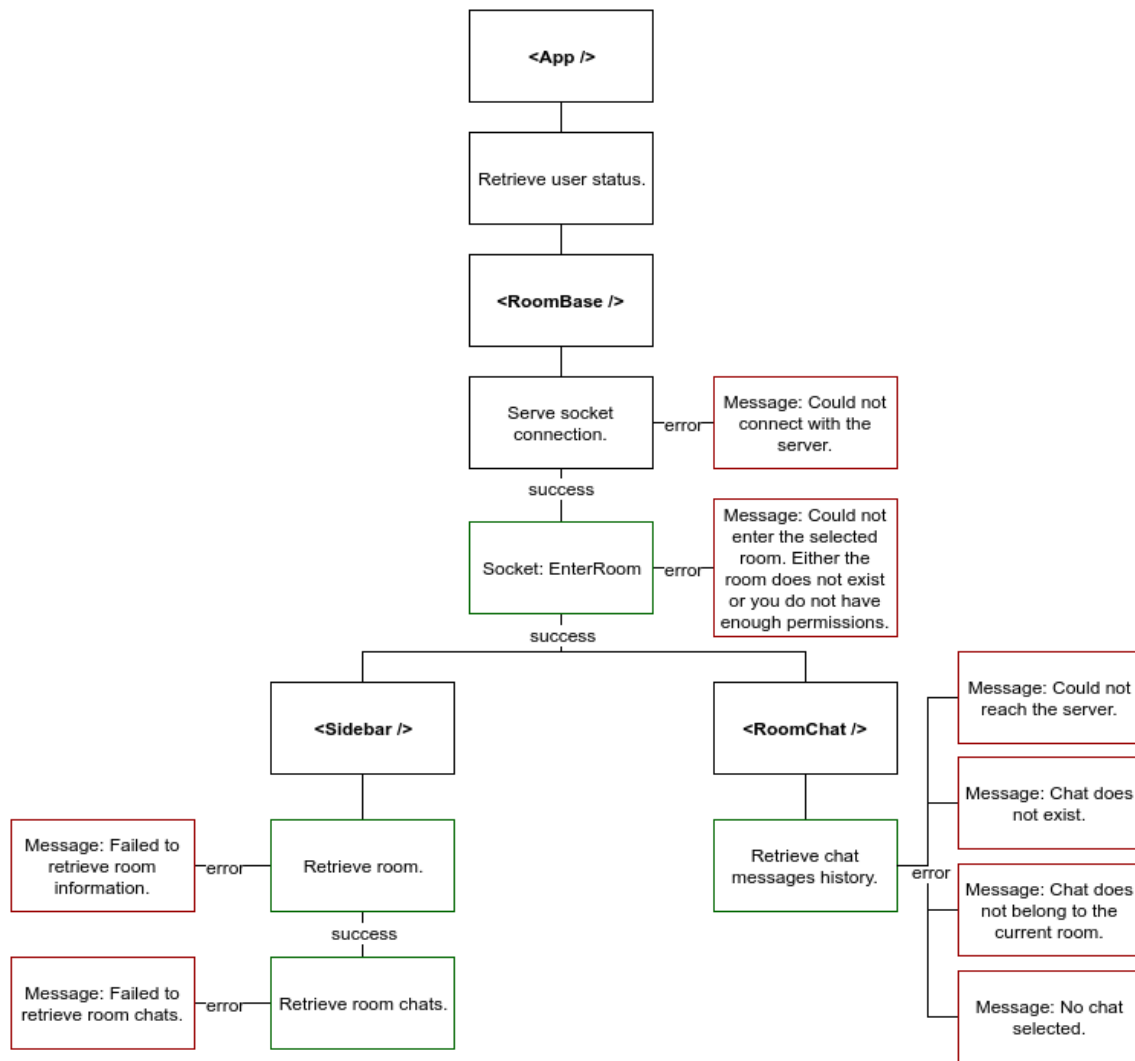
Figure 4.11: React Room activity diagram

Although this is probably the most critical aspect of our sites when it comes to performance (since it requires real time), we are also reusing the socket connection to handle the room and chat updates.

For example, a title or description update would be received immediately.

We are just taking advantage of the established socket connection to update some other components in real time, rather than checking for updates every once in a while which is what we would probably do otherwise.

4.2.6 Snippets

Next feature were snippets, what turned our generic messaging app into a messaging app for developers.

Figure 4.12: Visual result of the chat implementation

Since we were already delivering messages, we had a couple possibilities to integrate them into our application:
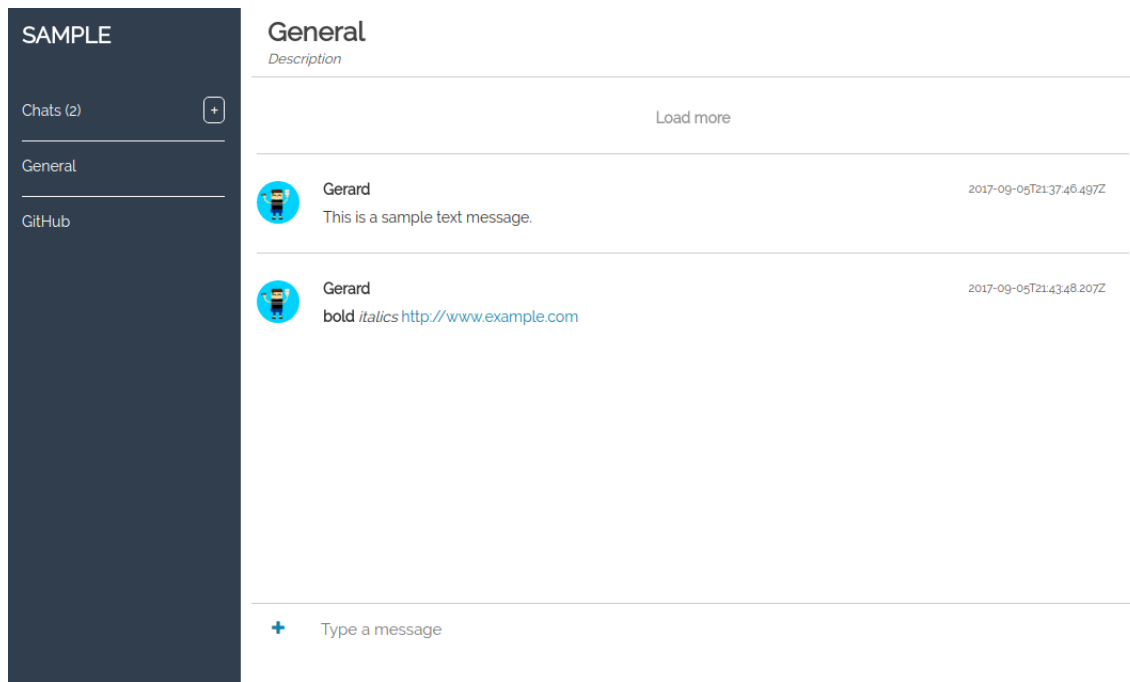
1. Expand the messages MarkDown support to include code snippets.

2. Create a different sort of message, that was code specific, that got a special treatment (just like Slack).

We went for the second one as we realized it had a few advantages:

1. From the UI side, it looks visually cleaner to have code separated from the text, and it's also easier to copy - paste.

2. It's easier to manage on both server and client sides.

3. Avoids using an escape string which can be broken unintentionally by parts of the code or natural text. (i.e. markdown "```" code wrapper might be used by someone who has no intention to paste or be part of some code snippets).

Apart from the advantages listed above, separating concepts will additionally make it easier to implement the highlighting feature later.

To do so, we create an additional field on the server Message schema, the `contentType`, that stores how the content message should be processed, which can either plain text or code.

```
content: String,
```

```
contentType: {
    type: String,
    enum: ['plain', 'code'],
    required: 'Content type is required',
},
```

Then, we added the language was the code snippet in onto the same schema. We could have also tried to guess the language the code snippet was written in, but by asking the user to indicate it, it makes the implementation a lot less complex and also we guarantee a successful syntax coloring.

```
contentTypeSpecifics: {
    language: {
        type: String,
        enum: ['plain', 'markdown', 'html', 'javascript', 'css']
    }
},
```

That was it for the server side, now it was time for the client to use that information in order to paint the code as beautiful as possible (making as it easy to read as if it was their editor).

The API was already providing us the language they were written it, so we just had to find out the easiest way to color them appropriately according to the language.

PrismJS[4] was a mature and very activate open-source library that focused on code formatting and also provided code highlighting which we are going to need later. So we did not think twice about plugging it in into our project.

```
(function() {
    const sayHello = function() {
        console.info('Hello');
    };

    sayHello();
})();
```

Figure 4.13: Prism JavaScript formatting.

By using PrismJS we had the displaying code snippets part sorted out, but the feature could not be set as complete just yet. The default text input was too small and uncomfortable to write code in (further problems listed below), and we

---

[4]PrismJS: https://github.com/PrismJS/prism

were expecting some users to write their own rather than just copy - paste it from somewhere else, so we needed an alternative prompt to submit these snippets.

1. It was too small.

2. The tab did not indent.

3. The enter key submitted the messages instead of going onto the next line.

4. New lines continued from the beginning instead of starting at the current indentation.

5. There were no line numbers.

That's when we decided to write the editor page as a dialog page[5] (see figure 4.14), so as to have a decent amount of screen space to write the code. In it users have a full page to write their code, choose its language and later they will be even able to highlight. When they are done they will be returned to the chat page.
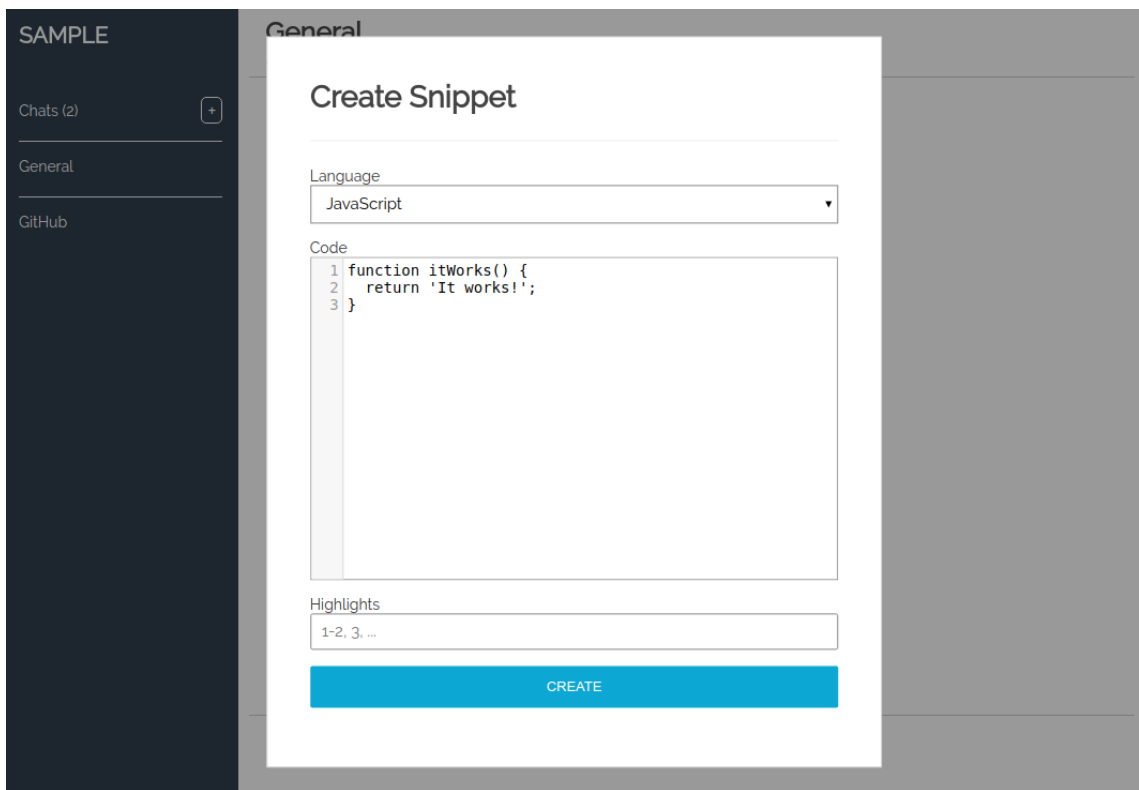

Figure 4.14: Create snippet form in a room chat

Since building an editor is quite a complex task, we used another library for this: CodeMirror[6]. It is also very active, open-source and maintained.

---

[5]Dialog page - a sort of temporal pop up page which overlaps the current one
[6]CodeMirror https://github.com/codemirror/codemirror

This whole process forced us to abstract Chat components even further. See Figure 4.15.
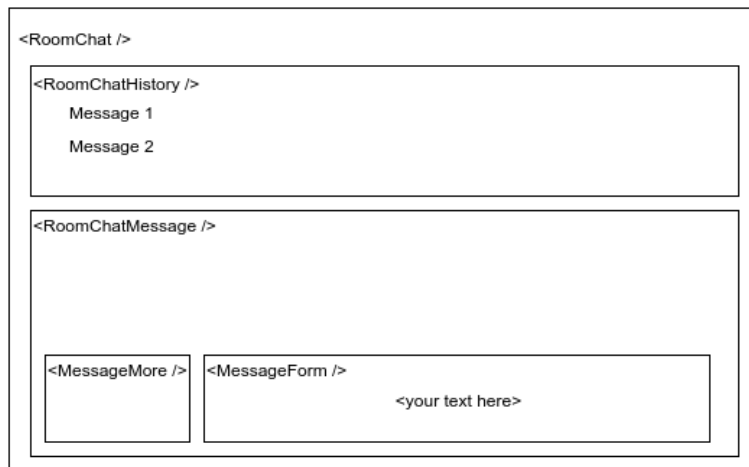

Figure 4.15: Chat React components overview

You might also wonder what we did with MarkDown's ability to style messages. While it is true that our intention is to support code snippets only through the tool detailed above, the MarkDown library we were already using to format messages already supporting basic code formatting out of the box, so we left it like it. However, users who are looking for a professional way to display their code will have to make use of our specific code snippets tool.

The result of using the snippets tool is shown in figure 4.16.

4.2.7 Sticky message

The sticky message is a message that is always displayed at the top of a chat to increase its visibility.

It is of special utility to developers to be able to discuss a particular snippet without having to scroll up and down the chat messages all the time. By using the sticky message they can discuss it while having the code visibility all the time.

Server side, our biggest dilemma was where to store these chat stickies. We thought of two possibilities:

- Create a new document for them (like we did with rooms, chats or messages).

- Keep the sticky information along with the chat database document.

Since we were only expecting a sticky message per chat, we went for the second one. We save the database from an additional query to another collection, and storing
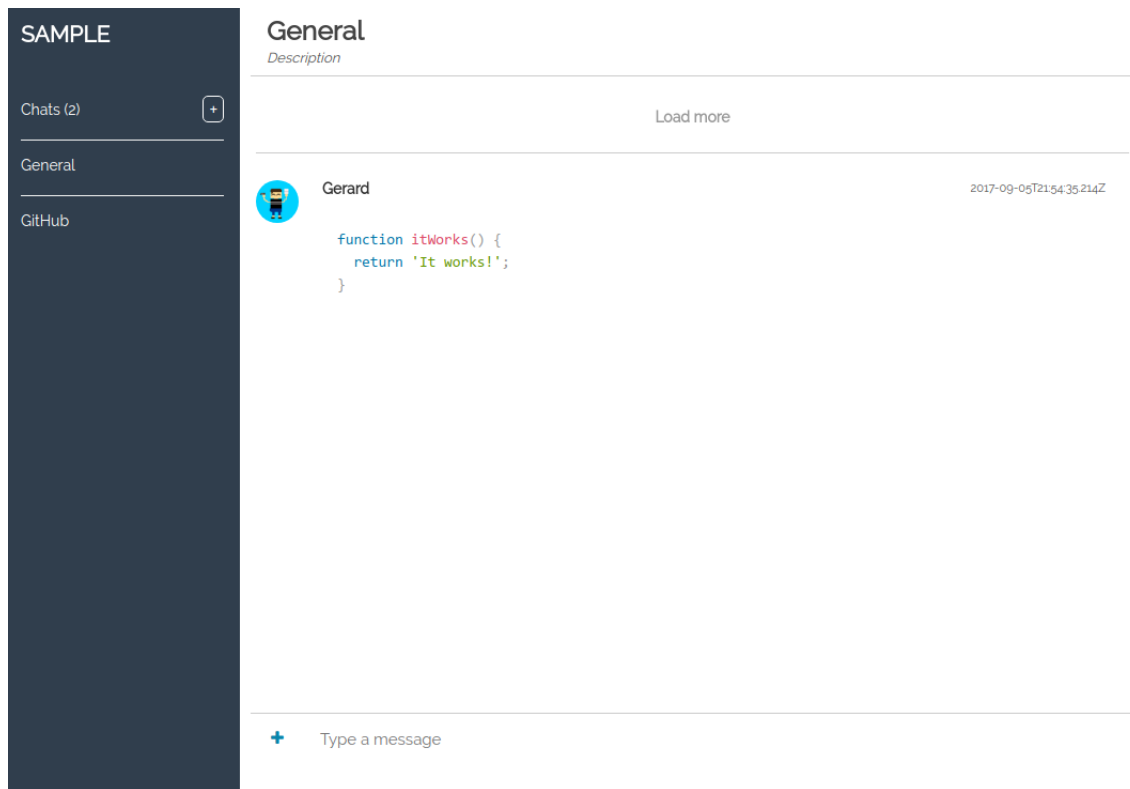
Figure 4.16: Snippet display on a chat

an only message into the chat document should not be a performance nor space issue.

Additionally, chat updates through WebSockets were already live on both the client and server (which we talked about in section 4.2.5), meaning that we had a generous part of the implementation working already.

On the client side, we first decided the position of the sticky. We had a couple options: stick it at the top of the chat or in a toggle panel at a side. Although a sticky message on top can often take a lot of the chat space, we thought it would be better for our specific audience, as it doesn't really help to have a snippet that is meant to be discussed hidden under a toggle panel. The result of the sticky placed on top is shown in figure 4.17.

When it comes to the implementation, we created an additional React component, ScrollContainer, to split messages responsibilities and avoid having a huge ChatHistory component.

The client side sends and receives sticky updates in real time through WebSockets, using the same event names as it would to update a chat, because, as we just mentioned when we talked about the server, the sticky message is part of the chat
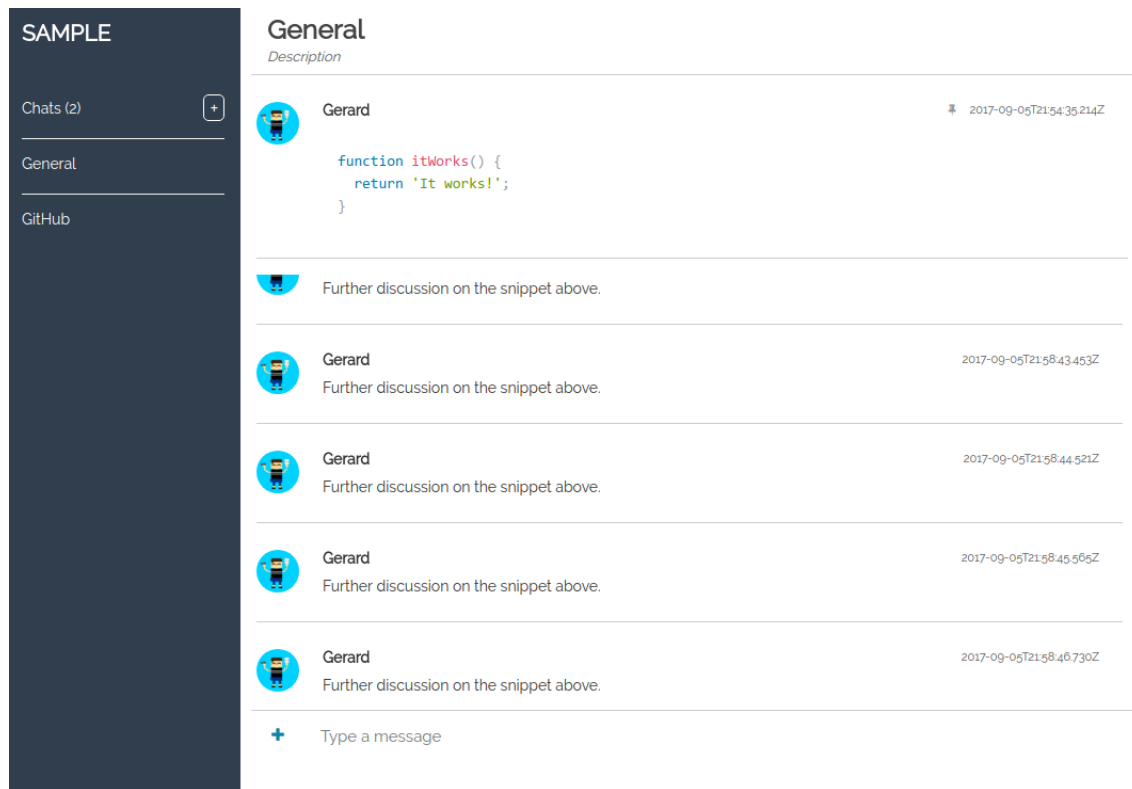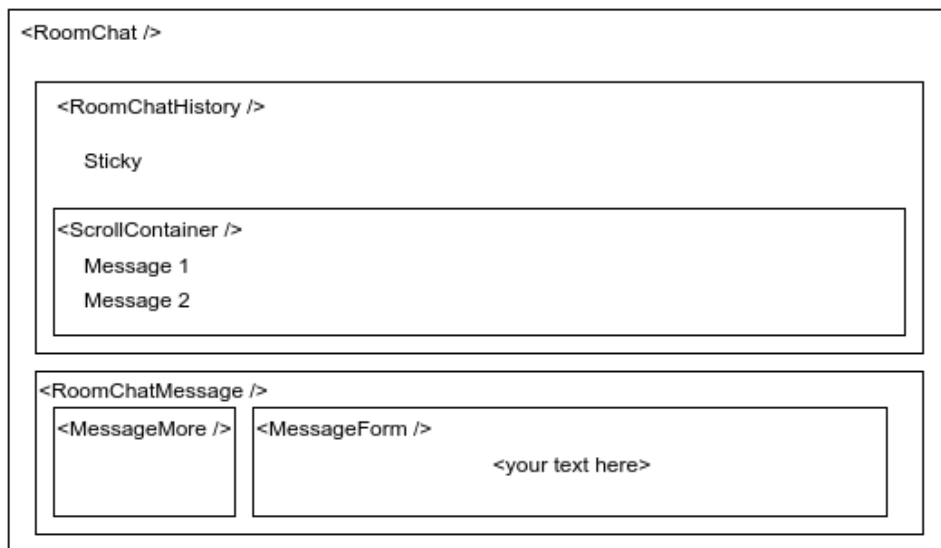
Figure 4.17: Sticky snippet with a discussion underneath

Figure 4.18: React components overview of chat stickies

document.

4.2.8   Snippet highlighting

When posting long snippets, it is often useful to be able to highlight certain bits of code which you want the other people to focus on, but without getting rid of the context.

As highlighted content has to be public, we had to create a new field in the message schema on the server side for it. We had no intention to process that value anyhow, so we just created a generic string field and left the format up to the client.

On the client, we were already expecting to use PrismJS for the highlighting stuff, as described in section 4.2.6. So, all we had to do is to follow their documentation to be able to show certain lines highlighted, such as the one displayed in figure 4.19.
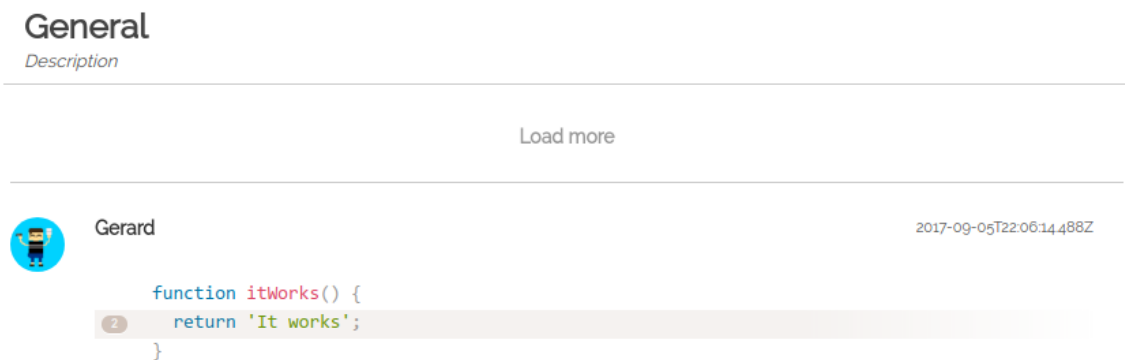
Figure 4.19: Snippet with a highlight

Additionally, we added an input field next to the snippet creation content, so that the user could select the lines they wanted.

When it comes to the lines format the client has to send in into the server, we made as similar as possible to do the values that PrismJS is accepting, so that the conversion was none or minimal.

It ended up having the following format:

```
1,10, 10-20, ...
```

Parts separated by a comma, dashing meaning highlight from this line to this other line.

When it comes to the UI, these highlights are shown with a very translucent yellow over the code letters.

4.2.9   GitHub activity

That was one of the most challenging parts of the project. We were looking forward
to obtaining the user's activity within a GitHub repository in real time. It would
consist in at least their pushes and pull requests.

We would then display this information as GitHub repository notifications on a chat
that is linked with that specific repository. This avoids having to start GitHub only
for the sake of checking updates and also enables some interaction with them.

When we started with that feature, there was a thing we knew for sure: we would
have to make use of their API (v3 at the time) to gather any sort of activity. Parsing
the content from their HTML was just not a viable idea.

The first approach we thought of was to send requests to their API to retrieve
the newest information for each of the repositories after a certain amount of time.
The number of repositories would be as many as they are linked to our platform
chats.

GitHub endpoints for commits, pulls, and issues:

- /repos/:owner/:repo/commits

- /repos/:owner/:repo/pulls

- /repos/:owner/:repo/issues

Doing it this way it would imply at least 3 API requests per repository every x
time. Not to say that X time would have to be relatively small due to the fact
that we pursue real time updates. Hence, it would become very expensive to scale:
a hundred linked repositories would mean thousands of API requests per minute.
What is more, lots of these requests would provide no value, because there might
have been no updates to that repository at that time.

Fortunately, GitHub API provides a better solution to this problem: WebHooks.

GitHub WebHooks allow subscribing to certain events that happen within a reposi-
tory or organization. As developers, we just have to have to a route ready to listen
for updates and they will send us all the new data in real time.

As you can see, WebHooks are the opposite from sending requests: we are now the
ones who are waiting for their requests. However, they still choose the format they
want to send the data in, and we will later have to process it before storing it into
our database.

That said, we had first of all to set up an endpoint on our Node for GitHub to send

the requests to:

`/webhooks/github`

That is when we faced a development obstacle. Our development setup was running under localhost (the loopback IP addresses), but GitHub servers would never be able to reach our `http://localhost/webhooks/github`.

We had to either forward our Node port on our router or use a tunnel. A tunnel to an external worldwide accessible host was a portable solution that works independently on whether you have control or not of the network, so we went for this one. Ngrok[7] and localtunnel[8] are both free services that allow us to do so.

Next, GitHub requires a one-time subscription for each repository that we want to watch (along with a list of things that we want to be notified of). So we created another URL for that: `/webhooks/github/subscribe`. While we could perfectly do that on the client, we opted to do it on the server on which we already have a user GitHub token stored to do so. Note that we require the user's GitHub token for this step, since all tokens have a restricted amount of requests, currently 5000/hour, and we could reach that limit easily by just using ours.

What the subscription URL does behind the scenes is to send GitHub a request with the user API key, informing of our WebHook URL and a list of things we want to notified of.

Once the subscription has been set, we have to make sure to properly validate the data that we are receiving on that WebHook endpoint:

The IP has to truly belong to GitHub

The GitHub WebHook endpoint is public. Hence, anyone can send in data. We have to make sure that the data comes from GitHub and not from anyone else.

There is also no authentication for GitHub, so we cannot whitelist any account just like we would do with our users.

The validation has to come down to the IP address or forwarded IP address when we are operating through a proxy (like we will in our production environment).

According to GitHub documentation, their current range of GitHub IPs which we can receive requests from is 192.30.252.0/22, so we drop any request that does not come from these range as we assume it has malicious intentions.

Request duplicates

---

[7]ngrok - `https://ngrok.com/`
[8]localtunnel - `https://localtunnel.github.io`

Either if messages got to their recipient correctly or not, GitHub offers the repository administrator to resend them at any time, and we do not want to end up having duplicated notifications if that occurred.

However, they do send a unique delivery identifier in every request, which we can use to verify whether we have processed that request already. So what we do is to store this delivery identifier in our database along with the data, and ignore processing any request that contains an identifier that we have already seen.

We can support their petition

GitHub can send us a wide range of data/events, which at first we are unlikely to be able to process appropriately.

Since GitHub names every request type, we can filter out the one we do not support. Currently, we support the following events: issue_comment, issues, pull_request and push.

Apart from the subscription stuff, we also provide an API endpoint for the client to be able to fetch all the activity that has reported to our API server on a GitHub repository:

`/webhooks/github/:repositoryUser/:repositoryName`

Client's logic is simple. First of all, it has to send a subscription request as soon as the user creates a chat linked to a GitHub repository. That is if the user indicate a repository when they create the chat.

Whenever a user is in one of these chats, it has to process the JSON-like activity coming from the server and display the data in a good-looking format.

All the logic about the subscription, processing GitHub requests and all the validation involved are already taken care by the API.

The activity is displayed on a dedicated vertical panel on the right-most side of the chat, which takes about 15-20% of the screen width (the result of which is shown in figure 4.20. Although we could have written the activity as chat messages, we preferred separating concepts to keep the chat cleaner.

Each activity message consists of few lines of very concise information about a specific event, which often contain references to GitHub to read the full original content.

When it comes to React, we had to divide the RoomChat once more to create the dedicated panel for the activity.
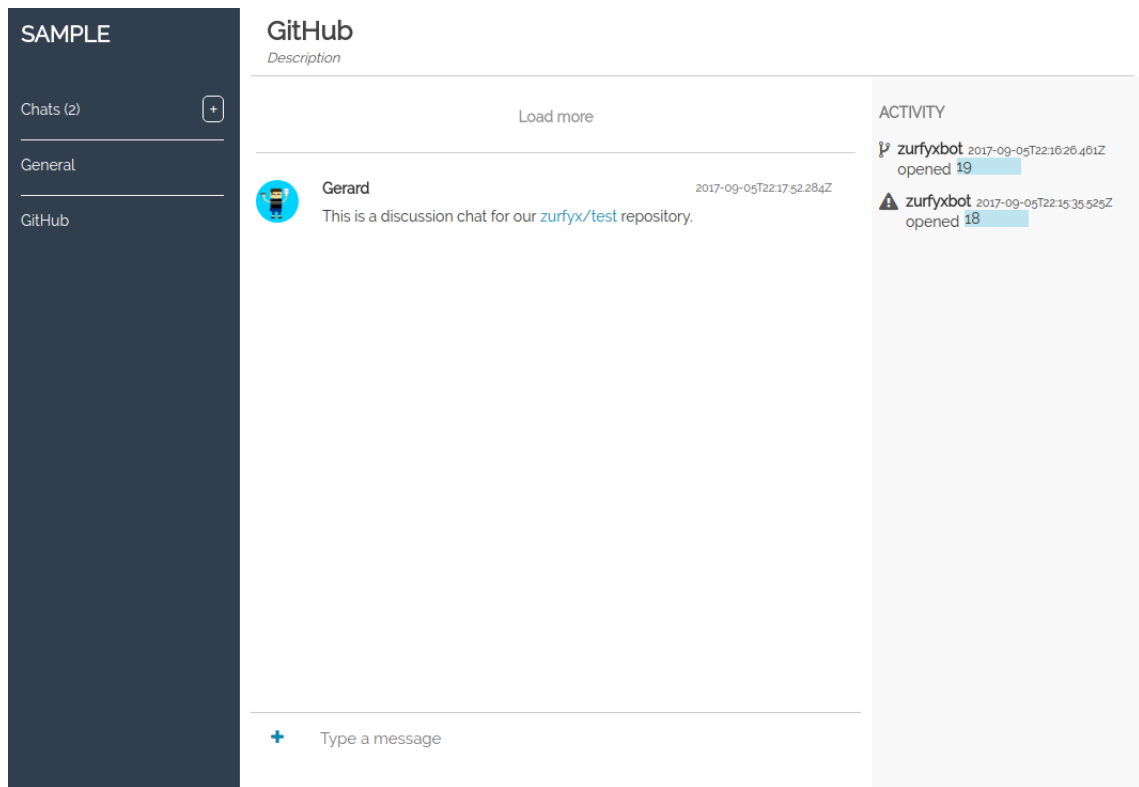
Figure 4.20: Result of the GitHub activity implementation
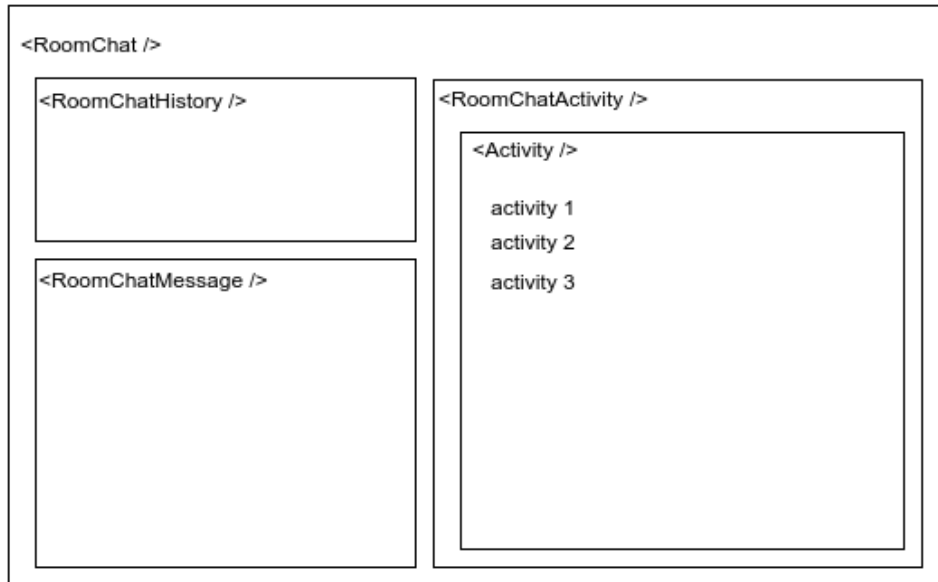

Figure 4.21: React components overview of chat activity

RoomChatActivity is the panel that holds these notifications, as well as making sure that the chat is subscribed to receiving new activity from the API.

Activity is the component that renders each activity object appropriately. Since each activity can contain different sorts of information, we have to process each of

them individually to make sure they are displayed in a concise understandable way (like we did on the server prior to store them).

4.3 Testing

Testing the source code, known as dynamic testing, is a kind of software testing technique using which the dynamic behavior of the code analysed[24].

By testing the source code we decrease the possibility of malfunctions after changing parts of it, even if they are not directly related to the modifications we recently committed.

Tests are a prerequisite for safe refactoring because it often implies modifying lots of code is being used many parts of the application.

Dependencies, parameters, and even function implementations might change, which may result in the application not working as it should, even if the application is compiling without errors.

There are dozens of test types to ensure that our application is fully working as expected. We will focus on Unitary, Integration and E2E tests which are the most common and that have proofed to be enough for the majority of projects.

That is not to say that other types of tests would not have been useful, but we had to balance between implementing new features and having enough tests. In fact, stress tests would have been great to verify that our application was able to handle a great number of members messaging themselves on the same chat at the same time.

A healthy tests distribution is said to be 70% unit, 20% integration, and 10% end-to-end[25].

4.3.1 Front end

The front end tests assess that the different components which are part of the client work as expected; so that it works all right for users when the API is up.

A React-Redux-Router app is set up by many components, but we focused our tests on Redux reducers, actions, handlers, and views.

Reducers
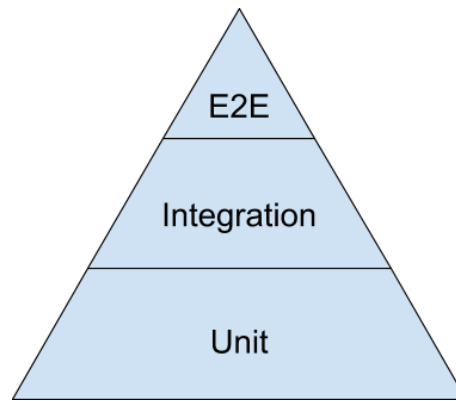
Figure 4.22: Right balance between unit, integration and end-to-end tests according to Google.

Redux, our in-memory client storage, is a reducer. It generates a new state after processing a recognized object.

What we wanted to test here is that given a certain object to the Redux reducer, the new generated (and returned) state is the one we were expecting.
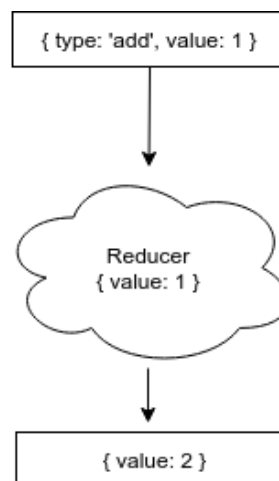
Figure 4.23: Reducer example.

In some cases, it is of our interest that the initial reducer state is empty, in some others we wanted the reducer to have a certain state in order to test edge cases.

Actions

Actions generate new objects for the reducer to process, which end up generating new Redux states.

The actions layer between the controllers and Redux is used to hide Redux inputs into simple functions.

Unit testing actions implied stubbing the dispatchers which sent the request to the reducer. We were then able to make sure that given certain parameters to the action

function, it would generate the expected object for the reducer.

Additionally, we could have set integration tests to make sure that given certain parameters to the action function, the ending reducer state would be correct.

Handlers

Our rendered views (the HTML that the user ends up seeing on their screens) have handlers of many kinds. The most common one is the click handler.

Enzyme, the JavaScript testing library for React, can simulate user actions by reproducing React content. Then we can spy or stub the handler function to make sure that it is being called.

Sometimes it is also useful to test the handler logic.

Views

Views end up being the HTML that the user sees on their screen.

It is often unnecessary to test them because the content in them is quite trivial (i.e. testing that a paragraph is indeed there, or that the button exists).

Only in some cases, it is worth testing them, such as when certain content should only display with a given state (i.e. show "sign in" button when the user is not signed in). Enzyme can help us with these sort of tests too.

E2E

Unlike the back end, we did not end up writing E2E tests for the client. But it was perfectly possible!

Tools such as PhantomJS (which is now about to be replaced for Chrome[26]) allow scripts or tests to execute themselves in a "real" browser, and navigate and read the content as a user would do. Hence, providing the most accurate feedback, even though they are more laborious to write than unit or integration tests.

4.3.2   Back end

Back end tests consist of making sure that the API correctly handles, stores and returns JSON responses to the user (through AJAX or WebSockets).

Our express architecture is divided into 4 parts: models, services, controllers, and router.

Each of them but the router which was too simple had their own set of unit tests. Apart from that we had E2E + Integration tests which to used to verify each of the

API endpoints, to make sure all the layers were working properly altogether.

During our tests, we were required to create spies, stubs, and mocks, which we explained during the analysis of the different tests.

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.

The following is a fragment of the unit test for the username validation:

```javascript
import { expect } from 'chai';
import { isUsername, isPassword } from '../validate';

describe('Model: User (validate)', () => {
  it('should be invalid if username length is not between 5-20', () => {
    expect(isUsername('x')).to.be.false;
    expect(isUsername('x'.repeat(21))).to.be.false;
    expect(isUsername('x'.repeat(5))).to.be.true;
  });
});
```

In some cases, we were required to replace implementations in order to ensure that the test was fully unit.

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[22] allowed us to make our router extremely simple, to the extend 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.

4.3.3   Integration + E2E

Lastly, we used a combination of Integration and E2E tests to verify each of the API endpoints. API endpoints are the different routes which the user can send requests to and often expect an answer in return (be it an error or a success message with an optional body).

In order to execute these tests, we have to run an instance of the server. Each of the tests will send one or more petitions to that server and will examine the response, simulating the behavior of a user (or our React client). For our AJAX tests we used `SuperAgent`, which is a library that runs on the `Node.js` platform and does pretty the same way as the native browser's `Fetch`.

These sort of tests were not as straightforward as Unit or Integration tests by their own. We had to launch an instance of the server and attach a fake instance (known as mocks) of MongoDB and Redis. That is because launching a server automatically runs these 2 databases/storages that will be later used to read/write data.

A Mock object is a simple implementation of a "real" object. They are used when we are not interested into testing the real implementation but only that our specific work is working properly. Mocks offer predictable and accurate responses which avoid having to depend on third party objects of which we do not have enough control, can have bugs or may not return predictable responses.

In our case, we used in-memory mocked storages to ensure that our real databases were never modified, and were easy to clean after each test.

One of our authentication E2E tests looks as follows:

```javascript
function signin(email = 'demo@example.com', password = 'password') {
  return new Promise((resolve, reject) => {
    request
      .post(`${server}/auth/signin`)
      .send({ email, password })
      .end((err, res) => {
        if (err) return reject(err);
        return resolve(res);
      });
  });
}


it('should log in with the right credentials', (done) => {
  chain
    .then(() => signin())
    .then(() => {
      request
        .get(`${server}/users/whoami`)
        .end((err, res) => {
          expect(res.status).to.equal(200);
          done();
        });
    });
});
```

In some cases, API responses are not enough to determine that the operation is being executed all right, and so we need to import specific parts of our source code to verify that the data is being stored as it should.

For example, in order to check that the user timestamps are being stored all right, we have to look up the user by querying the User's model directly after the AJAX

request has been completed.

Chapter 5

Phase 4: 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 DigitalOcean 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.

5.1   First attempt

Our first attempt was the classic one: on our target computer, we had to fetch the source code from our GitHub repository, install its dependencies (including databases), then boot up the server.

In our case, dependencies were easy to obtain. All the application modules referred to the NPM packages repository, so we could easily get them all installed by running "npm install" after the Node.js installation.

Databases were somewhat more complicated. They often have to be configured in a particular way to run OK in a production setup.

When we had all dependencies installed, we adjusted the production details to fit our server specifications, such as host or ports, and we started the application.

At that point, our application was live and accessible to anyone with Internet access.

Although the previous process neither took us too long nor it was especially difficult, two questions came to our mind during the process:

- What if we want to set up other servers with the same configuration?

- Do we have to repeat the whole process when we update the source code?

The fact that we did all deployment steps manually was indeed a problem. It was going to be very time-consuming in the long term run, and it would be even more if we ended up doing load balancing by adding more servers with a similar configuration.

Although we could have fixed both issues with a few lines of shell scripts, we thought it would be worth investing some time into finding a more sophisticated system/protocol.

Docker was a very attractive solution. Professional developers were using it in both their development and production environments, so we gave it a go.

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 Compose.

Docker Compose can manage several Docker containers. Each of them can communicate with each other and have a set of shared configuration while having their

runtime dependencies isolated from one and the other.

What really makes Docker Compose a better solution is the fact that many open-source projects, such as MongoDB or Redis, already offer their pre-configured Docker containers. Little to no custom configuration has to be done to a project that is already on the DockerHub, other than passing the appropriate command values when starting them.

Having both MongoDB and Redis containers on the DockerHub, we only had to craft our web one. We would next add them all on the same Docker Compose configuration so that they could communicate with each other.

The web Docker configuration file ended up being simple. All the specific Linux software that our project requires is a recent Node.js version. So what we did was the following: we extended an LTS official Node image and set it to copy all our source files inside the Docker container and install the NPM dependencies. Our web container also exposed port 3000 which is the one our Node.js starts by default, which we later proxied to 80 by using Docker Compose.

```
FROM node:boron

RUN mkdir -p /app
WORKDIR /app

COPY . /app
RUN npm install

EXPOSE 3000
CMD ["npm", "start"]
```

By using the previous Dockerfile code, we can generate an image which contains the latest version of our source code. With that image, we can build our project's Docker Compose configuration.

A Docker Compose file generally has a set of configurations for each of the images it handles, so we currently have one for the Web, one for MongoDB and one for Redis. Each of the services is responsible for executing the container with the appropriate environment variables, commands, port forwarding, and setting the appropriate dependencies.

Following, we have pasted a fragment of our base Docker Compose configuration file, which shows the web service configuration:

```
services:
```

```
web:
  image: zurfyx/chatapp:latest
  environment:
    - 'NODE_CONFIG={"database": {"data": {"host": "mongo"},
                                "session": {"host": "redis" }}}'
  ports:
    - "3000:3000"
  volumes:
    - /app/node_modules
    - .:/app
  depends_on:
    - mongo
    - redis
```

We are currently using two version of the docker-compose file. The newest one is the development one, as Docker does not only work well for production but also for development.

For the development one, we set our Docker configuration file to build a new image instead of downloading the latest one from the repository. This way we can test our image before publicly uploading it on the DockerHub.

At this point, we had one of the previous issues resolved; we were able to run a fully working instance of our application in any server without having to care about dependencies whatsoever.

We had just one issue left to fix: source code updates.

Our GitHub repository was already working with Continuous Integration services, which were taking care of the building and testing our application, and also notifying us when there was something which was not working as expected.

So we thought we could as well use these temporal Linux servers to update our remote server source code, and so that is what we did.

Using a small bash script, our CI, after successfully passing all tests, connects to our SSH server, pulls the latest GitHub source code, removes the previous Docker containers, re-downloads the new images and starts new containers by running the production Docker Compose configuration.

```
DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"
(
  cd "$DIR/.."
```

```
ssh $SSH_USERNAME@$SSH_HOSTNAME -o StrictHostKeyChecking=no <<-EOF
  cd $SSH_PROJECT_FOLDER
  git pull
  docker-compose pull
  docker-compose stop
  docker-compose rm -f
  docker-compose -f docker-compose.yml -f docker-compose.prod.yml up -d
EOF
)
```

Note that both our MongoDB and Redis data are safe from container deletions since we are storing them in a local folder outside the container all the time.

An alternative to the script above, which we found out later, would have been to use something like Rancher to handle the source code deployment. Rancher can manage one or more server configurations to deploy one or more application instances (what is known as Docker Swarm).

Chapter 6

Evaluation

6.1 Libraries / frameworks

Having worked with a few web development dependencies, some of which were new
to us, for a few months, we believe we are ready to review them, whether they met
our expectations, how difficult they were to learn, whether we would pick them again
in the future, ...

The main dependencies we are making use of are Express (the core framework of
our back end API), Socket.io (the WebSockets high-level library), React (the library
that powered our web client), and Redux (the in-memory client storage).

Although we are using many other libraries to power our chat application, such as
Validator or Redux-router, we will focus on the ones which were the most relevant
during the course of the project.

6.1.1 Express

Express is a minimalistic framework for web development, as they advertise them-
selves on their official site.

We chose Express in order to make routing easy with Node.js, since doing this by
ourselves with the http library can be quite cumbersome.

The Express framework by itself also offers the possibility to display content on
dynamically generated views, but we were not interested in that since we would be
outputting all our API content as JSON.

The framework resulted to be more powerful than we expected. Not only it made

routing simple, but also it handled user identification through the session cookie by itself.

Other than that it was handy to have a middleware system which allowed us to plug in validators as well as controllers to routes out of the box, which came really useful to get the project started, even though we eventually replaced most validators to become part of controllers, while we moved most of the logic to services[22].

6.1.2   Socket.io

Socket.io is a high-level library which hides the complexity behind WebSockets.

Socket.io allowed us to build the messaging communication system without having to dig into WebSockets specifics. While is it true that we may not have as much control of sockets as we would working directly with WebSockets, we did not require more than what the library offers.

Furthermore, Socket.io has an "acknowledgment" feature, which makes it possible to send a response to a request without any additional complexity, just as if it was an AJAX API. For example, with the same Socket.io request we are able to communicate the error (and even send a description of it) when a request has succeeded or failed.

Although the framework was really good, we disliked something about its design. Socket.io can have two or more different socket connections with the same ID (if they are coming from the same computer and session). We had to handle these same connections ourselves, to treat them as a unique user so that they shared the same session data and we had no duplicates when running different chat operations.

6.1.3   React

React is the current fastest growing client-side SPA[1] solution. It makes it possible to build fast and scalable user interfaces.

Although its learning curve is steep, we are convinced it was worth every minute we spent with it. React has a huge community behind it, and has plenty of documentation everywhere, not to say it is a really well-maintained software solution.

React by itself is just the View of the MVC architecture, but there are plenty of libraries available to complete the architecture and boilerplate pretty much all that

---

[1]SPA: Single Page Application

is needed for a project like ours (react-router as the router, Redux as the storage, ReactDOM as the renderer, redux-form as the forms system, etc.).

The fact that React generates HTML from JavaScript was a unique experience, and we think that it is a great solution to make the rendered solution more performant and less verbose to write.

React's tree structure (explained in section 3.4.2), also adopted by other frameworks such as Angular, makes the solution scalable and easy to follow.

As we were developing the project, we noticed how clever React's decisions were not just because they suited our project's needs, but also because other libraries and frameworks were adopting this "new" way to deal with the interface. In fact, if we were to start the project again now, we would be using Aphrodite instead of SASS. Aphrodite is a JavaScript library which generates inline CSS from JavaScript object styles.

6.1.4   Redux

Redux, the client side in-memory storage is our React's right hand. It is an in-memory storage for our client-side.

Its purpose is to make data sharing between different tree levels straightforward, rather than having to overload components with a lot of properties. Redux is especially useful when the node depth difference we want to share data within is 3 or more.

We have to say that it is not the only in-memory storage out there, and also it is not attached to React at all. We chose it on the first page because it was the most popular solution for React. Alternatives would have been MobX (an event-driven library) or NuclearJS.

Redux worked really well for our needs, and as our project got bigger we found out scaling the storage was possible by creating new storage modules.

What we liked the most about it is that it is possible to isolate it completely from the React's source code, thus making a better separation of concerns. It makes use of React properties to communicate storage changes, unlike MobX which requires of decorators to React classes' functions in order to notify storage data updates.

6.2   Methodology

We believe that an agile methodology was exactly what we needed to work on this project.

Back when we started, our project requirements were incomplete, mostly because we did not have a clear idea of what exactly we wanted to achieve, even after writing about the Ideal platform.

By using Scrum we were able to push new user stories into the priority list as the project was being developed.

Moreover, a two weeks sprint review worked well to analyze the work done during a moderate period of time and get professor's feedback about new features, as well as being a time to stop and think about new priorities.

Chapter 7

Future work

Although the application itself works well, much was learned during its development. For this reason, we wrote a list of possible improvements/changes, some of which are easy to execute, others might require rewriting a significant amount of the current source code.

Apart from that, the Ideal application (which we described in section 2.2) was too ambitious, which resulted in many features not being able to be implemented during the course of the project.

Express to Hapi

Express works well for small projects, it is easy to set up and you can have an API working within minutes.

However, it is very minimalistic. As the project gets bigger, you are forced to write much middleware code yourself, which does not only take time but it can lead to security risks if not properly tested.

Hapi is a more modern Node.js framework, with security in mind and designed to handle big loads.

Hapi by itself can handle things such as input validation, server-side caching, cookie-parsing or logging.

Although moving to Hapi is not a requirement, we believe it is a wise move since it would ease a lot of future work.

Move the whole API to an MVCS architecture

Back when we started our back end, we had models, simple controllers, and JSON responses as our views. Nonetheless, as the application grew, a few controllers logic

code got huge and repetitive. In some cases, we even needed to share logic between different application topics (i.e. authentication and chat rooms).

In order to fix this, we started by abstracting controllers into separate functions, but the separation of concerns was not clear. We had controllers, and controller "helpers".

That is when we decided to go for a not so well known architecture (Model-View-Controller-Service[27]), of which services have the critical logic and read and write from models[22].

Nevertheless, at the time we made the transition from MVC to MVCS the source code was already too big to make it all at once. Currently, around 40% of the API is still running with the old MVC architecture. Having it all running with MVCS would be beneficial for both the API and Sockets.

Node 7 over Babel

Ever since we started our application, we have been transpiling our JavaScript code with Babel for both the client and the Node.js server.

While our Node.js client works alright, some people argue against using Babel for the server[28]. That is because we are generating a different version of the source code than what we actually see (the transpiled code), which can eventually lead to some unexpected error.

To support this proposal even further, Node 7 has many of the ES6 features, including the latest async/await unveiled in ES2017, which makes Babel pretty much redundant.

Babel is still a need for the client though, as transpiling is required to support old browser versions that do not have the latest ES6 features.

Aphrodite as the CSS library

JavaScript generated CSS has proved to be less error prone than writing CSS by ourselves[29], even if using pre-processors such as SASS (which is the one that we used).

Aphrodite is not only the most well-known library that generates inline CSS styles, but also it was made specifically for React. After having used it in another project, we feel like it is a better choice than SASS in this case.

React server-side rendering

When working with SPA application, it is recommended to have the first page loaded server-side to avoid long loading times while the client is downloading at least the

main JS files.

Server-side rendering is not only beneficial for users who will have to wait for less than usual, but also for search engines which do not execute JavaScript before reading the content of the page. That is not the case with Google, even though it does penalize pages which take more than 2 seconds to display content on the screen.

Remaining features

The model platform, described in the "Features" section, had plenty of features. Many of them remain undone:

- Notifications

- Status

- Room roles

- File sharing

- Voice and videocalls

- Public API

- etc.

While our application already provides the basics to programmers who want to talk and share code themselves, having more of these model features done would probably attract the attention of more of them.

Chapter 8

License

The application source code is distributed on GitHub under the MIT license.

The URL of the source code is the following:

`https://github.com/zurfyx/chat`

This repository includes both the back end and front end parts. Anyone with the proper technical knowledge can run their own chat application instance(s) on the cloud without requiring any additional source code files by following the instructions placed on the README.

This document is licensed under the Creative Commons "Attribution-NonCommercial-NoDerivs 3.0 Spain".

Bibliography

[1] "Webrtc vs websockets." http://stackoverflow.com/a/18825175, sep 2013. Accessed: 2017-04-19.

[2] "Webrtc samples." https://webrtc.github.io/samples/, sep 2013. Accessed: 2017-04-19.

[3] "User stories: An agile introduction." http://www.agilemodeling.com/artifacts/userStory.htm. Accessed: 2016-10-20.

[4] "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.

[5] "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.

[6] "Why is nosql faster than sql?." http://softwareengineering.stackexchange.com/questions/175542/why-is-nosql-faster-than-sql, nov 2012. Accessed: 2016-10-21.

[7] "Why nosql." http://softwareengineering.stackexchange.com/questions/175542/why-is-nosql-faster-than-sql. Accessed: 2016-10-21.

[8] "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.

[9] "Composition vs inheritance - react." https://facebook.github.io/react/docs/composition-vs-inheritance.html. Accessed: 2016-12-18.

[10] "Mongodb manual 3.4." https://docs.mongodb.com/manual/reference/limits/. Accessed: 2017-08-30.

[11] "6 rules of thumb for mongodb schema design." `http://blog.mongodb.org/post/87200945828/6-rules-of-thumb-for-mongodb-schema-design-part-1`, may 2014. Accessed: 2016-10-18.

[12] "Scaling secret: Real-time chat." `https://medium.com/always-be-coding/scaling-secret-real-time-chat-d8589f8f0c9b#.m5jigxq6x`, may 2015. Accessed: 2016-10-18.

[13] "Analysis of json use cases." `https://blogs.oracle.com/xmlorb/entry/analysis_of_json_use_cases`, apr 2013. Accessed: 2016-11-08.

[14] "Crud cycle (create, read, update and delete cycle)." `http://searchdatamanagement.techtarget.com/definition/CRUD-cycle`. Accessed: 2016-11-08.

[15] "About native xmlhttp." `https://msdn.microsoft.com/en-us/library/ms537505(v=vs.85).aspx`. Accessed: 2016-12-18.

[16] "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.

[17] "Rfc 5789 - patch method for http." `https://tools.ietf.org/html/rfc5789`, mar 2010. Accessed: 2016-12-18.

[18] "Perfomance tips | google cloud platform." `https://cloud.google.com/storage/docs/json_api/v1/how-tos/performance#patch`. Accessed: 2016-12-18.

[19] "container vs component?." `https://github.com/reactjs/redux/issues/756#issuecomment-141683834`, sep 2015. Accessed: 2016-11-13.

[20] D. T. Andrew Hunt, *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Professional, first ed., october 1999.

[21] "Html5 websocket: A quantum leap in scalability for the web." `http://www.websocket.org/quantum.html`. Accessed: 2016-11-13.

[22] "Building a scalable node.js express app." `https://medium.com/@zurfyx/building-a-scalable-node-js-express-app-1be1a7134cfd`, feb 2017. Accessed: 2017-04-13.

[23] "Strategy design pattern." `https://sourcemaking.com/design_patterns/strategy`. Accessed: 2016-11-14.

[24] "Dynamic testing." `https://www.tutorialspoint.com/software_testing_dictionary/dynamic_testing.htm`, apr 2017. Accessed: 2017-04-03.

[25] "Just say no to more end-to-end tests." `https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html`, apr 2015. Accessed: 2017-04-03.

[26] "Stepping down as maintainer - phantomjs." `https://groups.google.com/forum/#!topic/phantomjs/9aI5d-LDuNE`, apr 2017. Accessed: 2017-04-13.

[27] "Mvcs - model view controller service." `http://stackoverflow.com/a/6216013/2013580`, jun 2011. Accessed: 2017-04-14.

[28] "Don't transpile javascript for node.js." `http://vancelucas.com/blog/dont-transpile-javascript-for-node-js/`, apr 2016. Accessed: 2017-04-14.

[29] "Css is fine, it's just really hard." `https://medium.com/@jdan/css-is-fine-its-just-really-hard-638da7a3dce0`, mar 2017. Accessed: 2017-04-14.