Building applications with NodeJS

Software development bachelor project

Zheni Dimitrova

Summer 2019

Table of contents

Bachelor project assignment contract

Software Development Top Up
School approved template

Student:

Zheni Nikolaeva Dimitrova
jeninidi@abv.bg
+45 71 505 280

Rampelyset 1
3,-lejl 67
4000 Roskilde

Project title:

Building a chat application with NodeJS

Basic project specifications/what this project offers :

- A functional chat application build with the NodeJS framework, using multiple libraries and packages in order to implement every functionality in an efficient and thoughtful way
- Users are able to pick a username and select a chat room where they can securely communicate with each other
- Relevant constraints in regards of message transfer handling, profane language and duplications
- Live version of the project deployed for public use
- Detailed project specifications can be found further in this report

Project Documentation:

- Problem formulation
- Introduction
- State of Art and Trends
- Specification of Requirements and Design
- Development and Implementation
- Conclusion
- Appendices

Assignment date: 31 July 2019

Project supervisor: Jarl Tuxen
jart@kea.dk

Presentation date: Not yet stated

Documentation and information sources are stated in References

Problem formulation

Throughout my experience working on several client projects during my internships, it caught my attention that various companies are using JavaScript as their to-go language when it comes to building web applications of any kind, for both back end and front end. One framework that particularly sparked my interest was NodeJS - as the name suggests, a JavaScript derived platform, used for developing back end - and whereas the idea for this project comes from - I wanted to try it out by myself and build an application. This bring me to the problem statement I formed:

NodeJS and its growing popularity amongst developers:

- **What does NodeJS have to offer?**
- **Is it as logical and seamless as portrayed?**
- **How to build an application with it?**

In order to find and support the answers to these questions I have made an in-depth research about the appliances, documentation and usage of NodeJS. With the help of multiple articles, materials and appendices I analysed the pro's and con's, best practices and coding conventions of the framework, while trying to use objective and legitimate sources.

Introduction

The purpose of this report is to provide the reader with detailed information about the scope, trends, requirements, specifications and final outcome of this project. Here you will learn about my motivation to chose this topic, project tasks and objectives, scope of the project and brief explanation of what to expect further in this report.

Motivation

The first thing that comes in hand is my motivation to chose this topic. In the world of technology and development we observe fast pace in change of trends, methodologies and workflow.

JavaScript as a programming language however, is stated to be keeping the title of most popular programming language in the last 5 years, according to a yearly developer survey[1] conducted by StackOverflow in 2017. The upper also shows that NodeJS is the most popular framework in the world, as stated by developers. Furthermore, JavaScript is also the most popular language on

[1] StackOverflow developer survey results

GitHub[2], with 2.3 million projects, followed by Python(1m) and Java(986k). This is the first reason I chose to research the topic.

The second reason is the one mentioned in my problem formulation - during my education I had the opportunity to work with different companies as an intern, where I observed the popularity of the language and framework in a real life enterprise environment. In regards to this I decided to conduct a research of my own and learn how to develop a project with it.

Project objectives

The expected outcome of this project is for me to develop a functioning chat application using NodeJS. The goal is that users will be able to pick a username and name their chatroom. Through sharing the name of the room with their targeted audience, they can give access to it and this is how other people can join. From there on, all users can send messages to each other. The application should also have additional features included to ensure no duplication of usernames and dynamic render of all current users in a chat room. It is essential that messages can not be accidentally delivered to the wrong chat room. Nice to have features are system notifications informing members about entering and leaving users, and also a function to share current location.

In order to form a proper project objective(s) I used the SMART technique:

- Specific - all objectives are stated in a clear and easy to understand manner
- Measurable - an approximate estimation of steps can be made
- Achievable - a project of such scope can be competed in the given time period
- Realistic - in regards to the conducted research, it is possible to develop based on the acquired knowledge
- Time-bound - there is an exact time period stated, allowing proper time management

Project tasks and scope of the project

[2] The State of the Octoverse 2017

In every development process there is a long path between the idea and the final result. At many times this process can be messy, and if there is no proper planning on hand, the final result can be negatively affected, or not be implemented at all. In order to accomplish the best case scenario, in the beginning of this project I made a definition of its scope. This helped me set defined goals and manage my time properly

The first thing to do after the idea was formed - chat app with NodeJS - was to define and prioritise all needed features and form them into clear tasks:

Essential tasks:

- Users can create chat rooms by setting username and room name
- Users can join chat rooms by setting username and inputting the name of already existing room
- There is a dynamic display of all current users in a chat room
- One user can be part of multiple chat rooms using the same username
- 2 users with the same username can not enter the same chat room.
- New members do not have access to previous messages, they can see only the ones that are sent after they joined.
- Data channels are secure and messages can not be sent across chat rooms.

Good to have tasks:

- Users are able to to share their current location with their chat-mates through link redirecting to maps application.
- When a new user joins a chat room, the rest of the participants receive an automatic message that a user(displayed with the username) has joined the chat. The same notification is generated also when someone leaves the chat.

These tasks were also helpful in defining the non-functional requirements of the project.

The next step in my project tasks was to settle on a specific design of the application. In my opinion it is important to settle on the basic GUI in the beginning of the planning alongside with the scope estimation in order to ensure that the user interface will be applicable to the given software tasks.

The first prototype consisted of 3 main pages. The home screen was designed to have 2 sections - one form for entering credentials and a second form displaying the currently active chat rooms. The second page was supposed to be the chat page, leading to the third page, where the real time rendering of current users in the char room would be displayed.

When the designs were complete, it was time for me to select the project environment and tools.

Based on the type of project, it is possible to be developed on all of the 3 main operating systems - Windows, Linux and macOS. My OS of choice is Windows, and this program is developed with Visual Studio Code IDE.

In order to work with NodeJS, 2 environment installations are needed - npm and node. Node is the actual NodeJS environment, followed by npm - "world's largest software library, package manager and installer"[3]. It is not possible to work with npm without installing node, therefore a full installation file can be downloaded from NodeJS official website.[4]

The tool for project management I chose is the Agile methodology. As Agile can be altered and adapted for the requirements of each project, it seemed like a proper solution to the given scope. During my research I also stumbled upon another technique intended to help the developer follow to the given scope and time of the project - overscoping. Based on both, I developed my workflow.

The implementation of the Agile methodology included separating the tasks into as small as possible sub-tasks, even if they seem minor and non time consuming. This is done in order to support the concept of overscoping, which is based on the idea to estimate 1-2 days for the smallest tasks and gradually increasing the time as the tasks get more complicated. As the following have already been destructured with the help of Agile, the developer is presented with a fluent, non-stressful working structure. The final outcome of this method

[3] According to www.w3schools.com

[4] See references

resulted in sufficient time management and completing the project within the time constraints.

After all tasks and subtasks were on hand, I had no difficulties in estimating the flow of the implementation process. I decided separate my work into 2 main steps - first to ensure that the functionality is implemented and functioning correctly, while keeping the design as basic as possible for test purposes; the actual UI design and styling followed after. A relatively detailed version control of the application can be found on my GitHub repository[5]

What exactly can be found in this report?

The following section is included in order to present the reader with 'brief description of all chapters'[6]

- Problem formulation - information about what subject this report and project are aiming to reflect upon
- Project objectives and scope - information about the goals of this project and what it aims to cover, alongside with motivation for choosing the subject
- State of art and trends - reflects on all of the main researched topics - NodeJS with NPM, Web sockets, Socket.io, alongside with interesting design packages and libraries
- Project requirements analysis - informs about the analysis of the project, functional and non-functional requirements, design solution and GUI tests; system architecture explanation
- Development and implementation - explains implementation steps for the programming part of the project.
- Conclusion - reflects on the entire project process, gives an adequate judgement about progress and future implementation ideas
- Appendices - contains relevant extra information about the project, coding samples, references user, installation manual, etc.

[5] See references

[6] As stated in the official KEA guide for writing BA report

State of art and trends

The following part of the report aims to reflect on the relevant parts from the conducted research and give in-depth information about the project topic, alongside with relevant packages and libraries used in the implementation of the software, as supported by it.

Starting off with NodeJS, as it is the main topic of this report and exploring npm that goes alongside with it, I found it interesting and relevant to also include information about web sockets and SocketIO (main concept in building a two way chat app), alongside with short info about 2 interesting node packages - MustacheJS and HandlebarsJS.

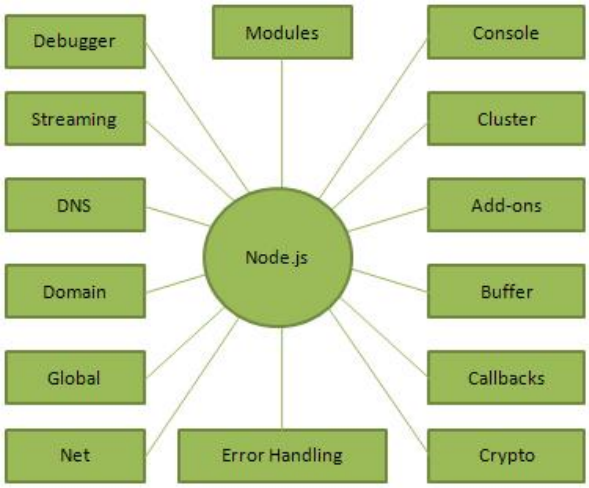
NodeJS

So far from this report you've learned that ,according to many studies and surveys, NodeJS has hit top popularity in the past 5 years. Here arises the question what exactly is it and what does it have to offer to the developers?

NodeJS is a runtime environment and framework used for developing server side JavaScript applications. It is asynchronous and event driven, which makes it the prefect solution when it comes to handling real-time applications. To further elaborate, NodeJS, being open-source and cross-platform, is mainly used for building the back end of services that need to communicate and transfer data with other client side applications with the help of various APIs. A lot of well-known and widely used services nowadays are built with NodeJS - E-bay, Yahoo!, PayPal, Netflix and Uber, just to name a few.

After we've learned what NodeJS is, we should ask the question of how exactly does it work, and isn't JavaScript mainly used for developing front end, not back end? The answer to the question is "No" - ever since Ryan Dahl found a way to outsource the Chrome V8 engine used for compiling the JavaScript code in the browser in 2009. With the help of this engine and C++, developers are now able to run the JS code as a standalone application named Node. This means that now Javascript has objects that allow manipulation of the filesystem. So how does it work? - The JS code is taken by the V8 engine, where it is being

converted into machine code. As machine code is low level code, it is very fast to execute as the computer doesn't need to do any interpreting in order to read it. However, it is worth mentioning that differences can be found between the JavaScript engines in each browser, therefore slight changes in the behaviour of the NodeJS code can be observed(and that's why we have browser web kits!) [7]



So far we've heard a few times the term 'asynchronous' in regards to NodeJS. In order to explain that properly, first we need to be familiar with the I/O concept(Input-Output). This basically translates into accessing any type of file/ data outside of your application and is connected with transferring of information between applications or server and client.

Most back end frameworks(i.e:Asp.Net) are synchronous(a.k.a blocking I/O) - they handle network or I/O requests by default. By doing so, if 1 request needs access to the filesystem, all other requests are kept pending until the current request is fully executed. This results in keeping the memory idle and increasing the response time. These frameworks can also work asynchronously, but in order to achieve that, the developer needs to manually input a lot of extra code. In that regard, NodeJS is different, as it is asynchronous(a.k.a non-blocking I/O) by default. It uses a single tread for the execution of all requests. It sends one request at a time, and in case a request needs access to the filesystem, the flow of requests keeps being executed. This is possible because each is handled

[7] see references for picture

independently and at the same time. This means no idle memory and fast data transfer.

To summarise all, NodeJS is a good software solution when it comes to I\O bound apps, as well as data intensive real time applications(a.k.a DIRT Apps), alongside with data streaming, single page applications and JSON API's. However, Node is NOT meant to be used for apps that require intensive CPU performance.

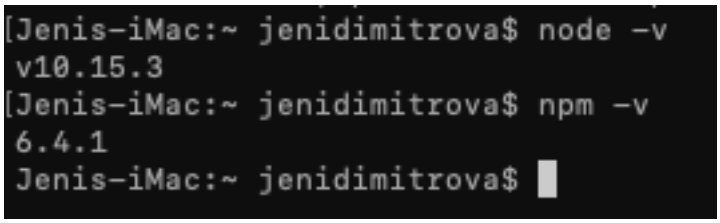
NPM

After learning about the basics of NodeJS, next one in line is NPM, in order to understand what the link between them is and what exactly NPM is.

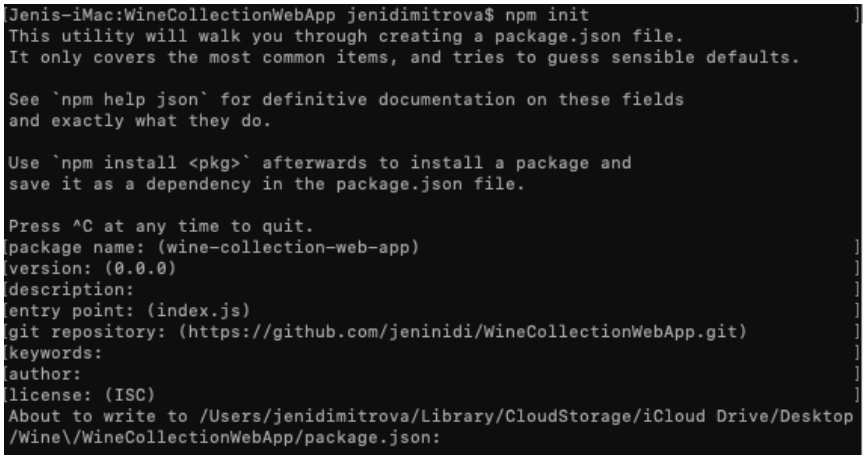
NPM(Node Package Manager) is known to be the world's largest software registry. In its basis it is an open source online repository for publishing NodeJS projects. You might ask, if it in fact acts like a platform for publishing code, how is it any different than GitHub for instance? The entire registry is hosted on the website https://www.npmjs.com and the purpose of the projects uploaded there is to distribute software solutions to mainstream coding problems - packages. Also, NPM is also a CLI (command line interface) through which developers can install node packages for their projects. Furthermore, NPM is also helpful in managing multiple versions of the code and its dependencies. As it is open-source, anyone can publish their code there as a private/public package.

So how do we get this amazing npm on our machines and in our project? From the simplified installation guide on their website, we learn that it comes along with NodeJS, and this should come as no surprise. We can simply download an installation file from Node's official website[8] and run it on our machine. Done! After the installation is complete, it is a good idea to check wether everything ran correctly, by checking the current version of node and npm:

[8] See references



After we've done this, we can fully incorporate npm into our projects - you can find a rich variety of packages and frameworks for both back end and front end - Angular, React, Express, Socket.io, jQuery and many more. On creation of the project, the developer has to run the 'npm init' command from the CLI, then answer the following questions for the metadata of the project(or simply set everything to default). When the process is finished, a new folder named 'node_modules' and a file named ' package.json ' will be added to the root directory. In that folder you can find the pre-installed packages and all newly added packages will also appear there. The file contains all node modules that are currently installed, alongside with specified version number and dependencies. [9]



The package installation is also simple - by navigating to the root folder of the project and running the 'npm i packageName' - it installs the package in ./ node_modules. Once this is done, you can simply require() it in your code and use it as if it was built-in.

Another interesting feature of node is that it can greatly reduce the size of a project, if it needed to be uploaded. With a proper package.json file, the project can be uploaded/downloaded everywhere, without needing to include the files from the node_modules folder(around 1000 for a small project). By running

[9] Screenshot from another project of mine, as npm init was already done for this one

'npm install' in the root of the project, all packages from package.json are downloaded locally.

Web Sockets and Socket.io

The next part of this research focuses on Web sockets, as this concept is the basis on which the code for this project is built.

Web sockets are a client/server technology used for 2 way communication. Soon after release thy became a standard for any type of application that requires duplex cross-platform connection. With Web sockets we can have immediate distribution of messages with little to no overhead, which results in barely any latency in the connection. So how are they so awesome and are they any different from a regular HTTP request?

With Web developers have access to long held single TCP socket connection. And we don't have to worry about the cross platform communication, as the TCP handshake deals with it for us, which makes them a reliable solution for largely scalable real-time applications. Here we have a technology that is uniquely different compared to HTTP. When we have a HTTP request, it constantly asks the server for new information in order to receive it. This is defined by the term long-polling or Comet and very often results in unnecessary latency - as every request that 'asks' the server for new information caries session cookies and metadata. Web sockets, however, don't need to ask the server for information, they can simply listen for it, as their type of connection is opened and there's no need for polling.[10]

Then how does a Web socket request work? At first, the client send a usual HTTP request to the server. This request contains a special Upgrade header,

[10] See references for picture

that informs the server that the client is attempting to make a Web socket handshake. If it is accepted by the server, the initial HTTP request is transformed into a socket connection that has the same TCP/IP. This makes the Web sockets a good solution for real-time apps, Internet of Things, Chat apps and in some cases even multiplier online games!

Socket.io is NodeJS's most famous Web socket library, thus the choice for this project. It is reliable, as it is not influenced by proxies, firewalls or anti-malware software. It also provides useful functionality in regard to auto-reconnection support: by default it will try to reconnect to the server, in case any crashes occur. This library also has a disconnection mechanism also known as 'heartbeat' - automatically notifying client or server when the other is dis/connected. However, it was the Namespaces that helped me settle on this choice.

With Socket.io developers can create Namespaces, where each of them is acting as a separate communication channel. In every Namespace one can define random channels - rooms- where sockets can join and leave. Additional functionality is added for broadcasting to every socket in a given room. This is how I also implemented the Rooms functionality in my project.

Additional libraries

After making the reader familiar with the ground coding concepts of this project, I believe it will be useful to also include some information about the interesting packages I stumbled upon(and implemented) for the design solution of the application.

The first one is MustacheJS, the library that I used for rendering most of my templates.It is the JavaScript version of the Mustache templating library and it's most interesting feature is that it is logic-less. This means no usage of if-statements, clauses or loops - only tags. MustacheJS provides both templates and views as the basis for dynamically rendered forms, as binding are done through json property objects - {{binding}}. What also contributes to my choice is the fact that it has support for multiple languages, therefore no separate templating system for the server side is needed.

Last but not least, another interesting library is HandlebarsJS, which is also based on the mustache templating language. Handlebars is a well-known templating engine, that is capable of separating the HTML generation from the rest of the JavaScript code. This results in writing of cleaner code. The library has an easy and seamless template syntax - a mixture of HTML, text and Handlebars expressions. The developers also can use the Handlebars 'helpers' - because pure JavaScript code can not be written inside the templates, we have functions that are accessible from the outside. This allows us to efficiently reuse and minimise code and helps us create complex dynamic templates.

Requirements specification and solution design

In this section of the report the reader can find information about the procedural flow and the steps that were taken to estimate and document the requirements of the project, analyse the intended user experience and provide clarity about the design solution for the chat application

Analysis

In the beginning of every software project, an adequate estimation of the targeted user group is vital for proper project planning, cost estimation and business planning.

As for chat applications and for projects of this type, there is a wide range of social groups that can be targeted - from every day communication between friends, to company level group chats. However, the targeted audience for this project are mainly youngsters, teenagers and young adults. This estimation is supported by the fact that people from this target group are most likely to benefit from the features this application has to offer. Chat rooms and location sharing are features that tend to be popular amongst the generally younger part of the population. Furthermore, researchers say[11] that young people nowadays tend to use a lot more swear words in comparison to their parents/grandparents

[11] See references

when they were the same age. Based on this, the application can have its biggest popularity in this target audience.

Another important step of the development process is to make an estimation of the intended user experience. But what is user experience? - in simpler words, how are the users supposed to interact with the application. An interesting concept to take into account here: we are the developers, not the intended users. What we might find useful and logical for an application from programming point of view, might not have the same perspective for the intended user, as they want to interact with the app in a different way(especially design-wise). When writing such estimation 3 important questions should be asked about the users - Why, What and How[12].

'Why' refers to why the users would use the application. As a chat app, its main purpose is to help establish communication between 2 or more people. One reason why our users would chose to use this app is due to its consideration towards user privacy and security. No trace of users' messages is left once the session is closed - we do not keep track of user conversations, as opposed to many popular messaging services nowadays. Users can also allow single use of location service when they want to send a location message - the application is not tracing nor saving it permanently. Another reason is the fact that users have fully anonymous communication - they do not need to verify identity or register, nor is email address or any other personal information required. As long as users chose unique name for their chat room and share its name only with the people they want to chat with, no intercepting users can be expected.

'What' refers to what users do when they interact with the application. In order to use the app, users have to navigate to the web page of the application - https://zheni-d-kea-chat-app.herokuapp.com/ . On the main page the users are presented with an entering form where they chose their unique username and chat name. This username is not permanent or saved, so for every time the app is used, different username can be set. Same goes with the chat room name, once the session is closed, the same room name can be used for another session. After the credentials are in place, users have to click on the 'Join' button, which will automatically redirect them to their new chat room. Once inside the chat

[12] See references for article about the concept

room, users are greeted with a system 'Welcome' message with timestamp. On the left side of the window users are able to see the name of the chatroom alongside with the usernames of all currently active users. In order to send a message, users have to type the message in the corresponding field at the bottom of the page and click the 'Send' button. Afterwards the message will be shown in the chat window alongside with the username of the sender and a timestamp. Important thing to note is that tis message will be distributed to all currently active users. In case someone wants to share where they are at the moment, users can click the 'Send location' button. The API will detect the current location of the sender and deliver a clickable message 'My current location', again displayed with timestamp and username. All chat participants can click the message and will be redirected to google maps page containing a pin of the sent location. The same sender can not send another location message before the first one is delivered. If users want to leave the chat room, they have to simply close the web page in the browser. In this case, the remaining active users in the room will receive a system message informing them which user has left and the dynamic render of active users in the left part of the window will be automatically updated. Same automatic message is generated also when someone joins the chat. It could happen that someone closes their browser by mistake and in such situation users can simply navigate back to the webpage, chose username and enter the same room name. On click of the 'Join' button they will be redirected to the same chat room. However, this entry will be treated as a new one, as the application does not keep track of their users. This means that the re-joined users will not have access to their previous messages. As profane language is not tolerated, if a user writes a message containing swearwords and clicks the 'Send' button, the message will disappear from the typing bar, but it will not be distributed to the chat.

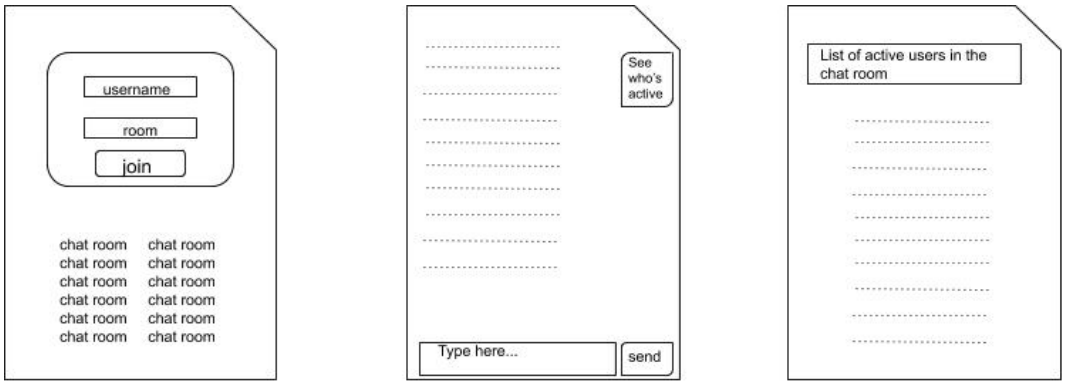
The last question is 'How' - this refers to the design decisions that were taken in order to achieve the best user experience.

Application design

In the project contract in the beginning of the report I included a brief explanation of the designs. Here I intend to give the reader an in-depth

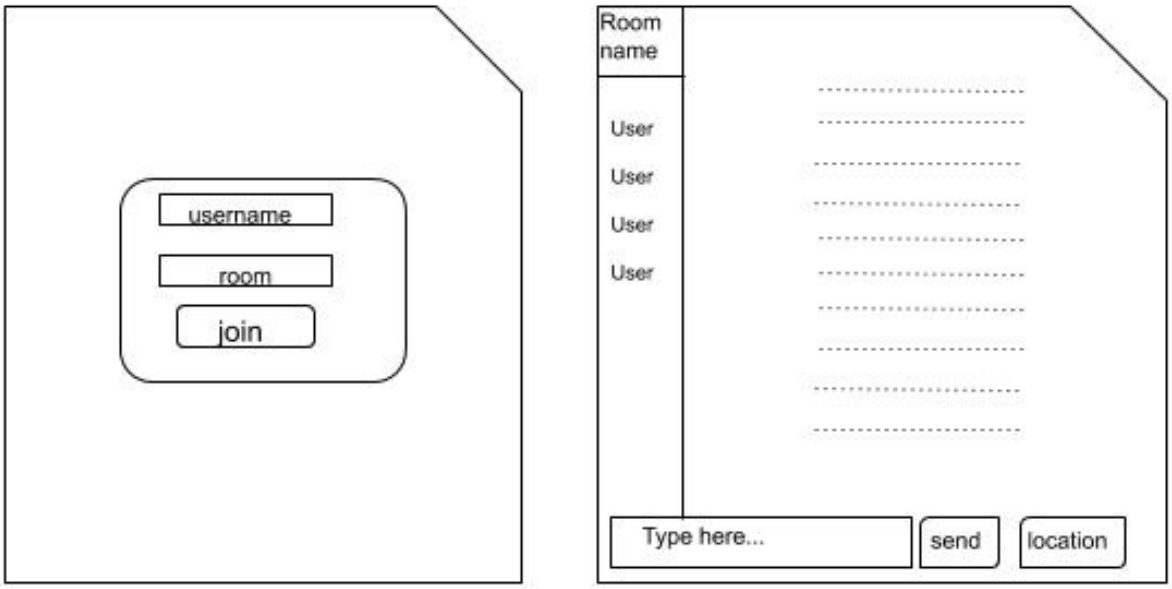
overview about the interface solution and explain why sometimes the first design is not always the best possible one.

In the beginning of the project, after settling on the idea and noting basic requirements, the next logical step to take was to design the project's GUI. The initial prototype was represented by 3 different pages - 'Home' , 'Chat' and 'Active users'. The home page here has 2 features - the entering form, where users can input username and room name alongside with 'Join' button and also, users are presented with a list of currently active chat rooms. The idea behind it was to increase the popularity of the application once it was released - when seeing all active chat rooms dynamically rendered, this would leave the impression that the app is used by many people. The second designated page was 'Chat', where the users were supposed to explicitly send messages to each other. It contained the life chat window where the messages from all users were displayed, accompanied by input field for typing and a 'Send' button for the messages. In the upper right corner of the page there's a button that redirects the user to the third page - made specifically for displaying a dynamic render of all active users in the chatroom. Sounds logical, right?



However, after I started coding the project, I realised a few flaws in this design. Even though the GUI seemed to provide a smooth user experience, it didn't quite meet the required functionality of the product. First of all, the home page had to undergo a re-design due to security reasons. If a list of all currently active chat tools was displayed there, anyone could see the names of those rooms and respectively join them, as we do not have a restriction about who joins those rooms - it is responsibility of the user to share the name with

whoever the conversation is intended for. Our users wouldn't benefit from having random people interfere with their private conversations, therefore the list was entirely removed from the homepage. The second page had a small inconsistency in its design - the app is supposed to allow users to share their location and no such functionality was represented in the design. The biggest change in the second version of the design happened when I realised that it is not as seamless for the user to go to another page just to see the currently active users, as I initially thought. This way the dynamic render of active members loses its purpose, as the users don't have constant access to it. From this came the decision to unite Chat page and Active users page. This way users are able to observe the immediate change of current users. Another extra implementation is to also include the name of the chat room above the rendered list - one user can be part of many groups and it might be confusing to see only usernames with no room name. From the second prototype came the actual application design:



After the implementation of the project was complete, I found it beneficial to include a Guerrilla usability testing in order to test the created interface. The application was distributed amongst 4 friends of mine, all of them being part of the targeted audience group.

Guerrilla usability test

What to test:

- Working design
- Ease of use
- Functionality

Who to do the test:

- Everyday people
- Targeted group of young people
- Students

Where to test:

- Home
- Place where testers have access to decent internet connection

How:

- Chose username and room name, click 'Join'
- Share room name with friend/s
- They have to pick username and type the same room name
- Type messages and click 'Send'
- Click 'Send location'
- Try to type message containing elicit language
- Try to leave and re-enter the chat room

Testers should pay attention to:

- Username and timestamp accuracy
- Username accuracy for joining and leaving members
- Dynamic render of active users list

Tester:	Interface	Ease of use	Functionality
Tester1	3	4	3
Tester2	2	3	4
Tester3	3	3	4
Tester4	3	4	3

Score it from 1-4, where 4 is best, and 1 is not at all.

Possible questions for the tester:

- Does the interface make sense?
- Is it easy/hard to navigate the application?
- Would you like to see changes? What type?

Notes (what input does the tester have to improvements):

Tester1: 'Interface is well made and logical, very easy to use. Only remark is why would you put a filer for swearing in a messenger app?

Tester2: 'Interface is working well on a laptop, but the version on mobile phone is a mess'

Tester3: 'The look is pretty basic, but everything works just fine and it's easy to understand'

Tester4: 'I set up my username to start with a capital letter, but it was displayed with a small one. Also, the google maps location is not perfectly accurate, but will do the trick'

Requirement specifications

Requirement specification are a vital part of every software project. They are created in order to inform developers and stakeholders about the exact expected functionality, alongside with time, cost, performance and usage estimations. Therefore, it is important to also say a few words about these requirements in this project.

Functional requirements

This section is dedicated to the functional requirements of the project. Here you can find the minimal functional requirements, ideas for future development, user stories and functional decomposition graph for user interactions.

Minimal functional requirements

- Users can pick username
- Users can create chat rooms
- Users can invite other people to join their chat room
- Users can exchange messages

- Users can share their location
- Users are not allowed to use profane language

Ideas for future implementation

- Users can exchange files
- Users can see all rooms they are currently active in
- Better mobile support
- Each username displays in a different color
- Re-joined users have access to previous messages

User stories

1) User user stories

”As a user I want to be able to create chat rooms”

Acceptance criteria:	User can create chat rooms
Preconditions:	User has access to the application
Basic flow:	User navigates to the main page
	User selects username
	User selects room name
	User clicks on ‘Join’
Success scenario:	User has joined a new room with corresponding name

”As a user I want to be able to join chat rooms”

Acceptance criteria:	User can join chat rooms
Preconditions:	User has access to the application
	Room with such name exists
	Username is not taken
Basic flow:	User navigates to the main page
	User selects username
	User selects the room name he wants to join
	User clicks on ‘Join’
Success scenario:	User has joined to room with corresponding name

”As a user I want to be able to send messages”

Acceptance criteria:	User can send messages
Preconditions:	User has access to the application
	User has joined at least one room
Basic flow:	User types message
	User clicks the ‘Send’ button
Success scenario:	User’s message has been successfully sent

”As a user I want to be able to see usernames for all sent messages”

Acceptance criteria:	User can see who sent each message
Preconditions:	User has access to the application
	User has joined at least one room
Basic flow:	User types message
	User clicks the ‘Send’ button
Success scenario:	Username is displayed next to the message and visible for everyone

”As a user I want to be able to share my location”

Acceptance criteria:	User can share location
Preconditions:	User has access to the application
	User has joined at least one room
Basic flow:	User clicks ‘Send location’ button
Success scenario:	User’s location has been successfully shared

2) Admin user stories

”As an admin, I want to inform my user if a new one has joined”

Acceptance criteria:	User can see all new participants
Preconditions:	Admin has access to the application
Basic flow:	New user joins the chat room

Success scenario: All users receive an admin message if a new user joins in stating 'username has joined'

"As an admin I want to inform my users when someone has left"

Acceptance criteria: User can see leaving participants
Preconditions: Admin has access to the application
Basic flow: Already existing user leaves the chat room
Success scenario: All users receive an admin message if a user leaves, stating 'username has left'

"As an admin I want to forbid my users from using profane language"

Acceptance criteria: User can't use profane language
Preconditions: User has access to the application
User has joined at least one group
Basic flow: User is typing message that contains profane language
User clicks 'Send' button
Success scenario: Profane message is not delivered

"As an admin I want to prevent 2 users with the same username to join 1 room"

Acceptance criteria: 1 room can not have users with identical usernames
Preconditions: User has access to the application
Basic flow: User types username that's already taken in the room
User types room name
User clicks on 'Join' button
Success scenario: User receives a notification informing that the username is already taken
User is redirected to the homepage

"As an admin I want to prevent users to further send location message if the first one is not delivered yet"

Acceptance criteria: User can't send another location message before the first one have been delivered

Preconditions: User has access to the application
User is part of at least one chat room

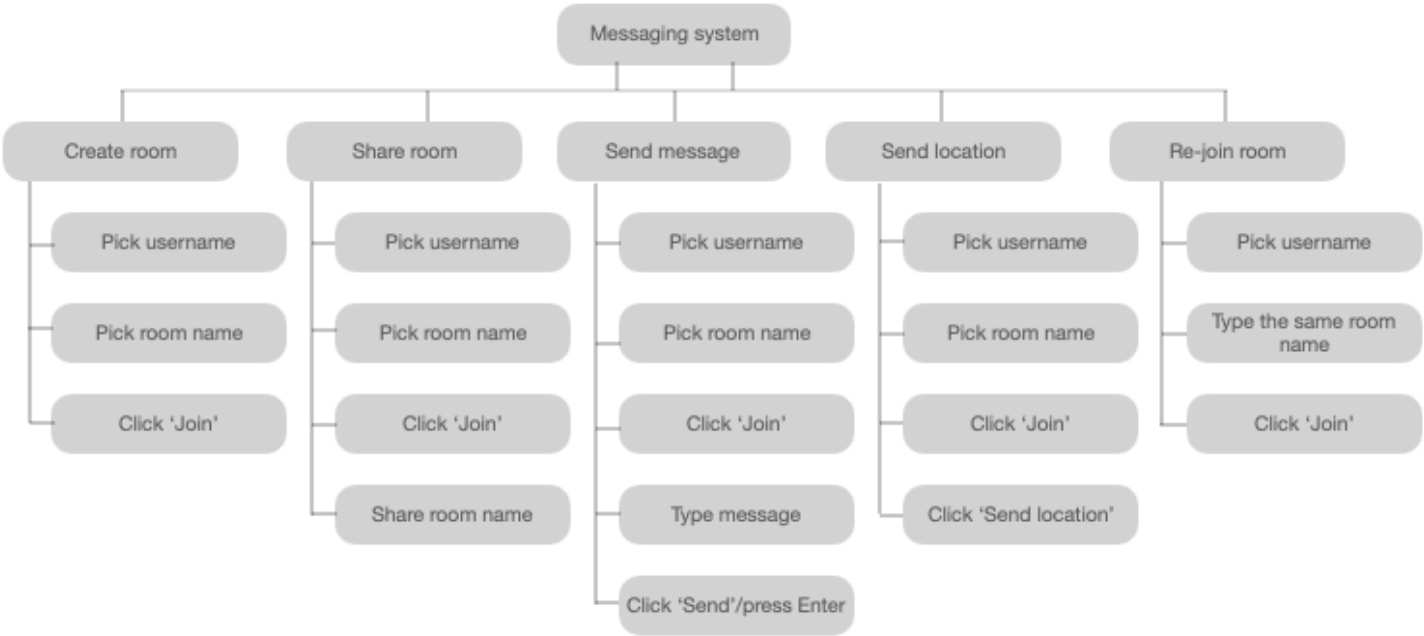
Basic flow: User clicks on 'Send location' button
User clicks immediately after again the 'Send location' button

Success scenario: After the first time the button is clicked, it will become disabled until the message is sent and it will be reenabled once the location message has been delivered

User stories are build after the INVEST[13] quality model.

Functional decomposition graph

Work Breakdown Structures'(a.k.a functional decomposition graphs) purpose is to take the interactions from highest level of complexity and break them down into as small sub task as possible. This allows us to make an independent analysis of each part.



[13] Independent, negotiable, valuable, estimable, small, testable

Non-functional requirements

The purpose of non-functional requirements are to set the exact behaviour the system should have and also to estimate the software constraints of the functionality. Here the reader can find relevant information about the system's quality attributes.

- Security - the technology with which this chat app is built ensures no tracing of data after the session has been closed. User information, messages and locations are not saved in the cloud, and no unique identifier is required in order for someone to use the messaging services. The only thing needed by the program are browser session cookies to establish the HTTP connection, and they are automatically deleted once the user closes the application. In case someone wants to re-join a chat room, a new session cookie is obtained.

- Reliability - in the matter of reliability, no major bugs are found in the tested parts of the interface and functionality - The application itself meets the reliability requirements. However, in case the user has a hardware/connection problem, this might influence the work of the application. Such failure is expected if the user's machine accidentally shuts down, or their browser application crashes.

- Performance - based on the guerrilla test done by 4 testers, functionality wise the program is operating as it is supposed to. Depending on the hardware of each users, the limit of properly functioning chat rooms at the same time may vary. The machine on which the application is built is a middle class average machine - i7 Skylake processor with 16 GB RAM - and the application started showing functional inconsistencies at around 600 rooms. The deployed cloud version shows availability of around 1500 simultaneous connections. As Socket.io has very low processing demand, the majority of users are not supposed to experience any troubles.

- Availability - as the application is live, the production version of the project is pushed to the Heroku cloud service. However, in case any changes are needed, developers are going to work locally on the development version, test their

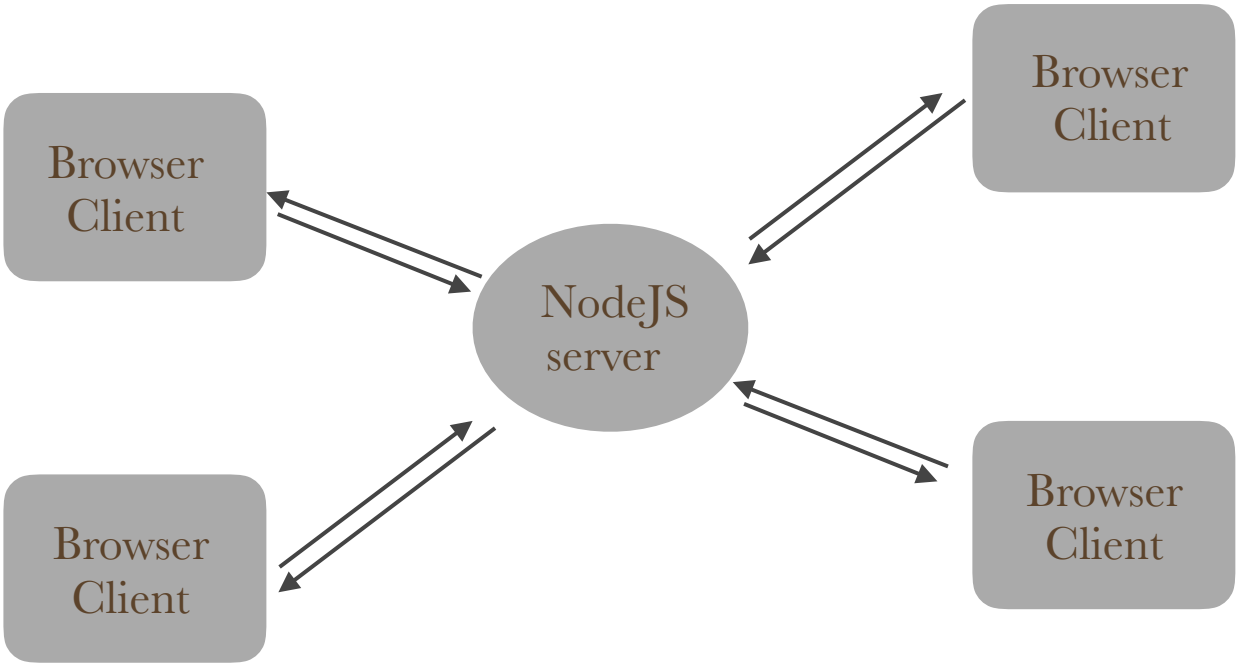
work and then merge the code on GitHub. During this process users will not experience any changes in the program. Once the new features are implemented, tested and approved, then the develop version will be pushed to the cloud service. In such case the application will be inaccessible for a time period of 30 second to 1 minute, based on the hardware of the machine doing the push. After that users will be able to navigate to the application in the same way as before and benefit from its new features

- Scalability - systems of such type are very easily scalable. If running locally, scalability depends on the ability of the user to upgrade their hardware. The live version's scalability depends on the cloud service of choice.

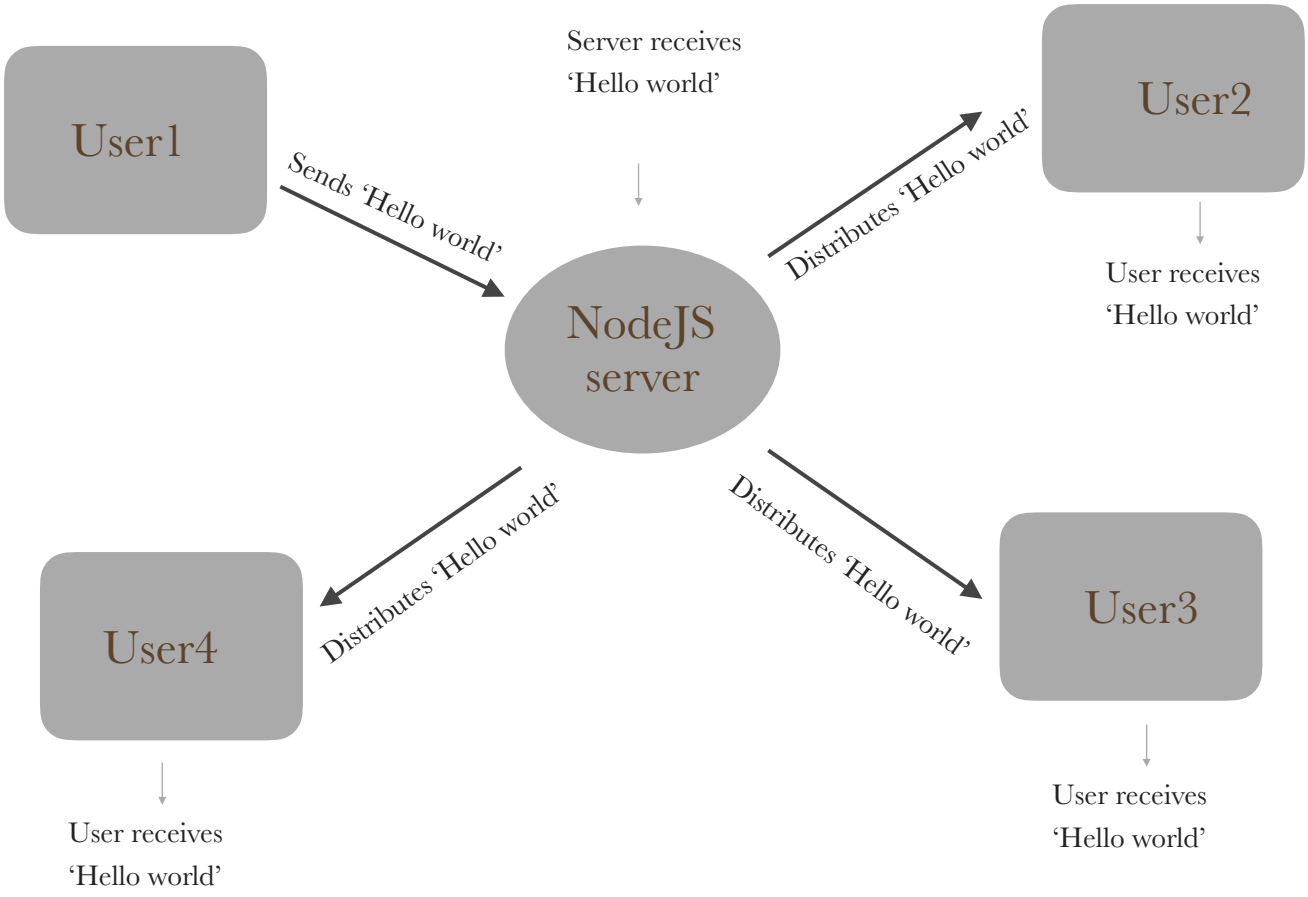
System architecture

This section is intended to make the reader familiar with the core architecture of the project.

In the base of our application we have a NodeJS server. Clients can connect to that server and a full duplex connection is established. This means that both client and server are able to communicate with each other at all times. Clients connect to the server and stay connected for as long as they need to. In our case this means that client and server can constantly exchange messages between each other - the connection goes either Client → Server or Server → Client.



Now let's look at the way the messages are being handled. When a Client wants to post a message to the chat room, it sends it to the server, and after that the server distributes this message to all other clients. As a user typer a message and clicks 'Send'/presses Enter, we have Client to Server communication. When this message is distributed to all other users in the chat room, we have Server to Client communication. And this is how the message flow between users is being handled:



Development and implementation

The last technical section of this report aims to provide detailed information about the implementation process of this application. The report contains short bits of code to illustrate the basic concept, full code snippets can be found in the Appendix

Earlier in this report was mentioned that the concept based on which the work tasks were separated is also known as overscoping - breaking off all major implementations into as small as possible features, so that developers can obtain an in-depth understanding of the task at hand and manage to stay within the time constraints of the project. Here the reader can learn about the step-by-step process of building this chat application.

First steps

First off I started by installing Socket.io in the root folder of the project, as most of the important functionality needs to be implemented through the web socket's functionality. Next step was to set up the express.js server, as Socket.io server requires to be set up in a slightly different way compared to a regular one. In order to do so I had to use the npm http library, so I can configure a new instance of the socket with it, and by doing so, my application was able to support web sockets. There I configured my server to constantly listen for new connections and log information in the console in case new connection was established:

index.js

```
1   const path = require('path')
2   const http = require('http')
3   const express = require('express')
4   const socketio = require('socket.io')
5
6   const app = express()
7   const server = http.createServer(app)
8   const io = socketio(server)
9
10  const port = process.env.PORT || 3000
11
12
13  io.on('connection', (socket) => {
14      console.log('New WebSocket connection')
```

In result I got notified every time a new user joined:

```
> chat-app@1.0.0 dev c:\Users\ZheniNikolaevaDimitr\Desktop\Chat app\chat-app
> nodemon src/index.js

[nodemon] 1.18.10
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node src/index.js`
Server is up and running on port 3000!
New WebSocket connection
```

Setting up events

After that I wanted to ensure that part of the core functionality is included, so I implemented a greeting 'Welcome' message, that was supposed to be automatically generated and displayed every time a new user joined in. This was the basic client-server set up - client attempts to connect to the server, server listens for the connection and sends back an approval when connected

```
33          socket.emit('message', generateMessage('Admin','Welcome!'))
```

Next step was to create the basic chat form containing an input field for typing and a send button and set id properties for both. These ids' were later binded to the listener of the submit event, so they can actually display the data. Important step here was to set up a function that prevents the browser from automatically refreshing, as this event would request a new socket connection every time it occurred:

```
$messageForm.addEventListener('submit', (e) => {
    e.preventDefault()
```

As a follow up I also set up a callback function for emitting events, in order for the server to be able to distribute the messages.

Broadcasting events and sharing location

After that inside index.js I set up a broadcasting event for informing the users when someone joins or leaves the chat. The goal here was for these messages to be send to all other users except the one that was joining/leaving. Good thing is

that Socket.io provides the developers with such pre-included basic functionality - ‘connection’ and ‘disconnect’:

```
//built in event
socket.on('disconnect', () => {
    const user = removeUser(socket.id)
```

When all of this was in place, I began the implementation of the ‘Send location’ feature. I did that with the help of a geolocation API. It worked in such way that it was using the client side javascript to fetch the location and sent it to the server:

```
navigator.geolocation.getCurrentPosition((position) => {
    socket.emit('sendLocation', {
        latitude: position.coords.latitude,
        longtitude: position.coords.longitude
    }, () => {
        //enable
        $sendLocationButton.removeAttribute('disabled')

        console.log('Location shared! ')
    })
```

Inside the UI form I created a ‘Send location’ button that was binded to the client, were the geolocation API was declared. On click of that button, users receive a prompt in the browser asking them if they want to share their location. If they allow it, the client takes their current longitude and latitude and sends it to the server. This data was at that time displayed in the console(later rendered as a clickable link with pin on a map)

Message approvals

Next step of the implementation was to implement message approvals. With minor changes to the client side I managed to target the value of each individual message in order to ensure it has reached the server:

```
const message = e.target.elements.message.value

socket.emit('sendMessage', message, (error) => {

    console.log('Message delivered!')
})
```

Inside the server I added a callback function to make the server send back a confirmation for delivery once the message was received :

```
socket.on('sendMessage', (message, callback) => {

    io.to(user.room).emit('message', generateMessage(user.username, message))
    callback('Delivered!')
})
```

The profane language filter is also a type of message approval - developers can add acknowledgements that check if a certain condition is met and based on the met criteria, the server can interact with the messages in multiple ways. The implemented language package is the npm 'bad-words' library. With it implemented, the server runs each message through the word registry and in case of a message containing swear words, the message is being rejected by the server.

```
if (filter.isProfane(message)) {
    return callback('Profanity is not allowed')
}
```

For the next part of the implementation I installed the mustache.js templating library from npm. Interesting feature about it is that developers can put html forms inside a script tag. This allows us to make logic - less bindings. This is how the messages are rendered and displayed:

```
<script id="message-template" type="text/html">
    <div class="message">
        <p>
            <span class="message-name">{{username}}</span>
            <span class="message-meta">{{createdAt}}</span>
        </p>
        <p>{{createdAt}} - {{message}}</p>
    </div>
</script>
```

Styling the application with dynamic forms

For this step of the development process I organised the div tags into proper forms with corresponding classes. Next followed the CSS styling, alongside with the implementation of the sidebar, which would later be used for displaying rendered lists of users. After adding the basic outline, I created a separate html page called chat.html. There I transferred everything from index.html. After refactoring of the code, chat.html contained the forms for rendering the actual chat, while index.html acted as the home 'Join' page for the application. There I implemented a form with 2 input fields - for username and chat room, followed by a 'Join' button. On click of that button the information from the form was submitted to the client, then sent to the server and the user was redirected to the corresponding chat room. Following, the chat room generates the message that new user has joined.

```
socket.emit('join', { username, room }, (error) => {
    if (error) {
        alert(error)
        location.href = '/'
    }
})
```

Handling of users

Another important implementation had to be taken into consideration so I could ensure that users are redirected to the right chat room. I did so by targeting the socket id and the room name followed by a call to the addUser() method. I modified the join method from index.js and I put the sendMessage()

method inside of it with the purpose to check and target the correct chat room for every message:

```
socket.on('join', ({ username, room }, callback) => {
    const { error, user } = addUser({ id: socket.id, username, room })

    if (error) {
        return callback(error)
    }

    socket.join(user.room)
```

This was followed by adding the dynamically rendered user lists method. In users.js I created a function that fetches all current users in the targeted room. In order to implement such list I had to follow up on 2 events: joining users and leaving users. Therefore the form is binded and triggered by those 2 methods[14] . Inside of chat.html I created another form inside a script tag for the sidebar UI. The name of the room is dynamically binded to always display the correct name, as it might get confusing when handling a load of rooms and usernames. Inside an unordered list tag I created an array of objects binded to all current users in a chat room. Inside of it I declared a list item the username was taken from the chat room and a binding was made:

```
<script id="sidebar-template" type="text/html">
    <h2 class="room-title">{{room}}</h2>
    <h3 class="list-title">Users</h3>
    <ul class="users">
        {{#users}}
            <li>{{username}}</li>
        {{/users}}
    </ul>
</script>
```

[14] See appendix

Deploying the application

First step of the deployment was to set up proper git.ignore file so I ensure that the node modules will not be included and pushed to the cloud. As my code was already hosted on GitHub, I only had to ensure that all changes are saved, merged and pushed properly, and that no bugs were experienced. After that I had to install Heroku on my machine - by simple navigation to the official website of the platform, I picked my operating system and downloaded and ran an installer. After the installation was complete, I had to include heroku in the environmental variables inside Window in order to allow it to access the filesystem of the OS.

After that I navigated to the root folder of the project and inside the terminal I ran the following command: ‘ heroku create ’ + the name of my application. The follow up needed command was ‘git push heroku master’ - which gets the code, pushed it to the Heroku cloud and attempt to deploy and run the application. In case of proper implementation, code should be up and accessible by anyone. Here you can see the project:

https://zheni-d-kea-chat-app.herokuapp.com/

Conclusion

Finally, in the end of this report it is appropriate to reflect on the entire development process for the project.

First, I would like to focus on the things that went well in this project. From the beginning of the project planning I had a solid idea on what I want to develop. This made it easy for me to estimate the proper requirements of the project and do an in-depth research of the topics. I am happy that I managed to learn so much new ways of implementation alongside obtaining a large amount of theoretical knowledge about the topic of NodeJS and Web sockets. In the end of tis project, I feel confident enough to provide an answer to the problems this development process aimed to shed light on - What does NodeJS has to offer and is it as logical and seamless as portrayed? The answer is yes - with Node and

JavaScript, we, the developers, have access to the world's biggest registry of software solutions - NPM - open source and free for use. NodeJS allows us to build complex applications without wasting our time on trivial implementations, which allows us to focus on the unique important features of each of our applications. In my opinion this is one of the main reasons Node and JavaScript managed to keep top position in a number of surveys and GitHub analyses for the previous 5 years!

However, no software product exists, whose implementation went smoothly, absolutely according to plan and does not need anything to be upgraded or fixed on it. Same goes for my project as well - there are some inconsistencies in my code and I believe a more experienced programmer could do it in much shorter and efficient way. Problems do occur in it from time to time, but there is always space for improvement in the world of programming! I also have a few ideas about the future implementation of the project - as one of my user experience tester mentioned, the mobile version of the application is a mess - it does not scale properly on different mobile screens, users experience problems with the scrolling, the resolution appears to be too small to be comfortable. Functionality wise it's working, but with such bad graphical design, no one would like to use it. Some improvements on the functionality can also be welcomed - it would be a good idea to allow users to create profiles in the future and set passwords for their char rooms. Another important feature of a chat app is to also allow users to exchange files and use the camera to send photos to each other.

Thank you for the time taken to read my report.

Appendix

Here the reader can find relevant information about the project, alongside with installation guide and research references.

References

This is a list of all references used for this project - for research purposes, coding challenges and proper structure of a software project report

Developer survey results - published by GitHub in 2017
The State of the Octoverse - published by GitHub in 2017
Should You Learn Node.js in 2018? (Spoiler: yes. Find out why we do it) - published by Olga Trqd in 2018
How to write a good project specification - published by Alexey Semeney in 2017
Use project objectives to structure the project and validate success published by Tom Mochal in 2015
How to define the scope of a project - published by Sarah, technical project manager in 2018
How to effectively scope your software projects - published by Angela Zhang in 2017
https://www.npmjs.com/get-npm - Official website of NPM - footnote 4
https://github.com/jeninidi/chat-app - GitHub repository for this project - footnote 5
Node.js Introduction - published by W3schools.com
What exactly is Node.js? - published by Priyesh Patel in 2018
NodeJS introduction - published by tutorialspoint.com
https://nodejs.org/en/ - Official website of NodeJS
What is NodeJS and Why You need to learn it - published by Michael Henderson in 2019
Which language wins in terms of salary/demand - published by Carl Joseph in 2014
Overview of Blocking vs Non-Blocking - published on NodeJS official website
https://docs.npmjs.com/about-npm/ - Official documentation for NPM
A Beginner's Guide to npm — the Node Package Manager - published by Peter Dierx in 2019
What is npm? A basic introduction for beginners - published by Fitrana A. In 2019
What are WebSockets? - published on Pusher official website
What are Web Sockets? - published by Dominik Tarnowski in 2017

What are WebSockets?? - published by Yamunda Navada in 2018
Introduction to WebSockets - published by Shivam Mishra in 2018
Socket.io: let's go to real time! - published by Mathieu Nebra in 2018
https://socket.io/docs/ - official documentation on Socket.io
Javascript Templating Language and Engine— Mustache.js with Node and Express - published by Sherlynn Tan in 2019
Creating HTML Templates with Mustache.js - published by Rakhita Nimesh in 2015
Handlebars and Node.js - published by Raja Raghav in 2018
https://handlebarsjs.com - official website of Handlebars.js
Learn Handlebars in 10 Minutes or Less - published on tutorialzine.com in 2015
Node.js Query String Module - published by W3schools.com
Focus On The Intended Users Of Your Systems – And That's Not You - published by Alan Zeichick in 2018
User Experience (UX) Design - Why What How model
https://github.com/drewww/socket.io-benchmarking - published by Drew Harry in 2017

Installation guide

Here the reader has access to step by step instructions on how to run the application:

1. Download the .zip file and extract the files
    1. Alternatively, write the following command in the console:
       `https://github.com/jeninidi/chat-app.git`

2. Navigate to the root folder of the project, open a terminal and install the needed node packages with the following command:
       `npm install`

3. After all package installation is done, run the application by typing the following command in the terminal:
       `npm run dev`

4. Open your browser on http://localhost:3000
    4. Or alternatively, visit https://zheni-d-kea-chat-app.herokuapp.com

5. Enjoy chatting with your friends!

Code snippets

1. 'Connection' code snippet

```
io.on('connection', (socket) => {
    console.log('New WebSocket connection')

    socket.on('join', ({ username, room }, callback) => {
        const { error, user } = addUser({ id: socket.id, username, room })

        if (error) {
            return callback(error)
        }

        socket.join(user.room)

        socket.emit('message', generateMessage('Admin','Welcome!'))
        socket.broadcast.to(user.room).emit('message',
        generateMessage('Admin',`${user.username} has joined!`))
        io.to(user.room).emit('roomData',{
            room: user.room,
            users: getUsersInRoom(user.room)
        })

        callback()
    })
```

2. 'Disconnect code snippet'

```
    //built in event
    socket.on('disconnect', () => {
        const user = removeUser(socket.id)

        if (user) {
            io.to(user.room).emit('message', generateMessage('Admin', `${user.username} has left!`))
            io.to(user.room).emit('roomData',{
                room: user.room,
                users: getUsersInRoom(user.room)
            })
        }
    })
})

server.listen(port, () => {
    console.log(`Server is up and running on port ${port}!`)
})
```

3. ‘Users.js’ code snippet

```
const removeUser = (id)=>{
    const index = users.findIndex((user)=>{
        return user.id === id
    })

    if (index !== -1) {
        //removing users from array by targeting id
       return users.splice(index, 1)[0]
    }
}

//get user by id
const getUser = (id)=>{
    return users.find((user)=> user.id === id)
}

//get user from room
const getUsersInRoom = (room)=>{
    room = room.trim().toLowerCase()
    return users.filter((user)=> user.room === room)
}

module.exports = {
    addUser,
    removeUser,
    getUser,
    getUsersInRoom
}
```

4. ‘Chat.html’ code snippet

```
30      <script id="message-template" type="text/html">
31          <div class="message">
32              <p>
33                  <span class="message-name">{{username}}</span>
34                  <span class="message-meta">{{createdAt}}</span>
35              </p>
36              <p>{{createdAt}} - {{message}}</p>
37          </div>
38      </script>
39
40      <script id="location-message-template" type="text/html">
41          <div class="message">
42              <p>
43                  <span class="message-name">{{username}}</span>
44                  <span class="message-meta">{{createdAt}}</span>
45              </p>
46
47              <p><a href="{{url}}" target='_blank'>My current location</a></p>
48          </div>
49      </script>
50
51      <script id="sidebar-template" type="text/html">
52          <h2 class="room-title">{{room}}</h2>
53          <h3 class="list-title">Users</h3>
54          <ul class="users">
55              {{#users}}
56                  <li>{{username}}</li>
57              {{/users}}
58          </ul>
59      </script>
```

5. ‘Chat.js’ code snippet

```
44
45  socket.on('message', (message) => {
46      console.log(message)
47      //loading the library
48      const html = Mustache.render(messageTemplate, {
49          username: message.username,
50          message: message.text,
51          createdAt: moment(message.createdAt).format('h:mm a')
52      })
53      //selecting where exacty to insert the new events
54      $messages.insertAdjacentHTML('beforeend', html)
55      autoscroll()
56  })
57
58  socket.on('locationMessage', (message) => {
59      console.log(message)
60      const html = Mustache.render(locationMessageTemplate, {
61          username: message.username,
62          url: message.url,
63          createdAt: moment(message.createdAt).format('h:mm a')
64      })
65      $messages.insertAdjacentHTML('beforeend', html)
66      autoscroll()
67  })
68
69  socket.on('roomData', ({ room, users }) => {
70      const html = Mustache.render(sidebarTemplate, {
71          room,
72          users
73      })
74      document.querySelector('#sidebar').innerHTML = html
75  })
76
77  $messageForm.addEventListener('submit', (e) => {
78      e.preventDefault()
```

6. Deployment

```
KEA+zhen0101@DESKTOP-6ABQOC6 MINGW64 ~/Desktop/Chat app (master)
$ heroku create zheni-d-kea-chat-app
Creating zheni-d-kea-chat-app... done
https://zheni-d-kea-chat-app.herokuapp.com/ | https://git.heroku.com/zheni-d-kea-chat-app.git

KEA+zhen0101@DESKTOP-6ABQOC6 MINGW64 ~/Desktop/Chat app (master)
$ git push heroku master
Enumerating objects: 837, done.
Counting objects: 100% (837/837), done.
Delta compression using up to 8 threads
Compressing objects: 100% (779/779), done.
Writing objects: 100% (837/837), 1016.52 KiB | 720.00 KiB/s, done.
Total 837 (delta 169), reused 0 (delta 0)
remote: Compressing source files... done.
remote: Building source:
remote:
remote: -----> Node.js app detected
remote:
remote: -----> Creating runtime environment
remote:
remote:        NPM_CONFIG_LOGLEVEL=error
remote:        NODE_ENV=production
remote:        NODE_MODULES_CACHE=true
remote:        NODE_VERBOSE=false
remote:
remote:
remote: -----> Build
remote:
remote: -----> Caching build
remote:        - node_modules
remote:
remote: -----> Pruning devDependencies
remote:        audited 200 packages in 0.999s
remote:        found 0 vulnerabilities
remote:
remote:
remote: -----> Build succeeded!
remote: -----> Discovering process types
remote:        Procfile declares types     -> (none)
remote:        Default types for buildpack -> web
remote:
remote: -----> Compressing...
remote:        Done: 20.4M
remote: -----> Launching...
remote:        Released v3
remote:        https://zheni-d-kea-chat-app.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/zheni-d-kea-chat-app.git
 * [new branch]      master -> master
```