

ATTESTATION

I understand the nature of plagiarism, and I am aware of the Institute's policy on this.

I certify that this dissertation reports original work by me during my Institute project except for the following:

- The API referenced by screenshots in section 4.2.3 were created using <http://apidocjs.com/>.
- The ERD represented by an image in 4.2.4 was created in Microsoft's Visio software.

Signature

Date

Project Document

**Jonathan Browne & Aaron Power
(N00113019 & N00121603)**

Project Title ECHO Web Communication Application

**Dissertation submitted in partial fulfilment for the
degree of BSc in Computing Multimedia
Systems/Web Engineering**

**Dun Laoghaire Institute of Art & Design
Technology**

ACKNOWLEDGEMENTS

We would like to thank our project supervisor Marion McDonnell for her valuable feedback and opinion throughout the academic year. Her weekly meetings gave us vital help in adjusting both the application and the final report to an acceptable state.

TABLE OF CONTENTS

CONTENTS

Attestation	1
Acknowledgements.....	3
Table of Contents.....	4
Table of Figures.....	7
Abstract.....	8
1 Introduction.....	9
1.1 Background and Context.....	9
1.2 Overview of Dissertation	9
2 Feasibility & Requirements.....	11
2.1 Requirements Analysis.....	11
2.1.1 User Requirements	11
2.1.2 Functional Requirements.....	13
2.1.3 Non-Functional Requirements.....	13
2.1.4 Technical Requirements	13
2.2 System Model.....	15
2.3 Feasibility	15
3 Research & Analysis	17
3.1 Systems Analysis.....	17
3.1.1 User Registration/Sign in	17
3.1.2 Add/View/Remove Friends.....	17
3.1.3 Call/Message Users	18
3.1.4 User Interface	19
3.2 Implementation Platform Requirements	21
3.2.1 JavaScript	21
3.2.2 Node.JS.....	21
3.2.3 Express.....	21
3.2.4 MongoDB	22

3.2.5	AngularJS	22
3.2.6	Gulp	23
3.2.7	WebRTC	23
3.2.8	Socket.IO	24
3.2.9	Jade & Sass.....	25
3.3	Prototype.....	26
4	Design	28
4.1	Client Side	28
4.1.1	Registration Screen.....	28
4.1.2	Login Screen.....	30
4.1.3	Main Screen	31
4.1.4	Call Screen	32
4.2	Server Side	32
4.2.1	NodeJS.....	32
4.2.2	Express.....	33
4.2.3	REST API	33
4.2.4	MongoDB	40
5	Implementation.....	42
5.1	Technology/Library Summary.....	42
5.1.1	NPM.....	42
5.1.2	Bower.....	43
5.1.3	Gulp	43
5.1.4	Gulp Plugins.....	45
5.2	Server Infrastructure	48
5.2.1	Technology/Library Summary.....	48
5.2.2	NodeJS.....	48
5.2.3	Express.....	49
5.2.4	Server Configuration	50
5.2.5	REST API	50
5.2.6	MongoDB	51
5.2.7	Socket.IO	52
5.2.8	Web RTC	52

5.3	User Interface.....	53
5.3.1	Screens.....	54
6	Testing.....	61
7	Conclusion.....	62
7.1	Evaluation	62
7.2	Future Work.....	62
8	References.....	64

TABLE OF FIGURES

Figure 2.1 - User Registration	11
Figure 2.2 - User Friends	12
Figure 2.3 - User Calling	12
Figure 2.4 - User account Creation	Error! Bookmark not defined.
Figure 2.5 - Home Screen Mockup	13
Figure 2.6 – System Overview	17
Figure 3.1- User Friend Options	18
Figure 3.2 - User Calling Options	18
Figure 3.3 - Splash Screen Mockup	19
Figure 3.4 - Register Screen Mockup	19
Figure 3.5 - Login Screen Mockup	20
Figure 3.6 - Home Screen Mockup	20
Figure 3.7 - User Registration	22
Figure 3.8 - User Calling	25
Figure 3.9 - Screen 1	26
Figure 3.10 - Screen 2	26
Figure 3.11 - Screen 3	27
Figure 3.12 - Screen 4	27
Figure 4.1 - Register Screen Second Version	28
Figure 4.2 - Form Validation Code Sample	Error! Bookmark not defined.
Figure 4.3 - Login Screen Second Version	30
Figure 4.4 - Main Screen Second Version	31
Figure 4.5 - Call Screen Mockup	32
Figure 4.6 - Get Users Friends	34
Figure 4.7 - User Sign in	35
Figure 4.8 - User Account Creation	36
Figure 4.9 - Update User Details	37
Figure 4.10 - Accept Friend Request	38
Figure 4.11 - Remove User from Friends List	39
Figure 4.12 - Database ERD	41
Figure 5.1 - Login Screen Final Version	54
Figure 5.2 - Register Screen Final Version	56
Figure 5.3 - First Dashboard Screen	57
Figure 5.4 - Dashboard View when Friend Clicked	58
Figure 5.5 - Dashboard Screen during Call	59
Figure 5.6 - Account Settings Screen	60

ABSTRACT

In this project the aim was to create a web application that allows users to call one another over the internet. The intention is to provide a browser based alternative to native web calling applications like Skype.

To achieve this aim a data driven single page application was built. The application uses only JavaScript, HTML and CSS and runs in the web browser which makes the app very quick. The intent was to use new technologies such as AngularJS and MongoDB to create this application.

1 INTRODUCTION

1.1 BACKGROUND AND CONTEXT

Currently a number of web based calling services exist. Many of these well-established services are used across the world for free and quick communication. However these services are built as native applications that are installed on your computer. The problem with this method for web calling is that native applications tend to be rather poorly optimized. These applications take up memory on the user's hard drive and can be quite taxing on the computer's hardware. Applications such as skype use a lot of power from a computer's CPU and volatile memory. This greatly limits the use of these applications on slower/older computers.

Our proposed alternative to these applications is a browser based web calling service. Running the application through the browser lightens the taxing cost on the computer when compared to other web calling services. This allows much older PC's access to powerful web calling. This also makes management of the application from the development side much easier. As the app is run in the browser, any updates can be handled from the development side, the user is not required to do anything, and updates are automatic.

Our application provides all the functions of a native web calling app but makes the web calling platform more accessible to users.

1.2 OVERVIEW OF DISSERTATION

Chapter 2 outlines the user, functional, and non-functional requirements of working on the project. Defining what the user will be able to perform within the project's system, and defining how the application should behave, and what is the main parts of the project, and what is tertiary to the goal of this project. And showcasing how the workload of the application is divided within the system between the technologies used, and talking about how feasible the project is in the time frame given by the college.

Chapter 3 will show the research taken into deciding the technologies that would be suitable for the project. It will also go into detail over how the systems laid out in Chapter 2, would be done within the abstract of using the technologies for the project. And finally providing a background on the technologies, and why they suit the project. Screen shots taken in November of 2014, will show a proof of concept application that only transmitted video, and audio across peers based on rooms.

Chapter 4 will be outlining how the two sides will designed both in a graphical mock up, and technically. Showing both, and early mockup of the project, and outlining how the

client handles user behavior. Outlining the details of the back-end (e.g. the relationship diagram of the database).

Chapter 5 will be examining how the project was developed. Detailing exactly how each technology was used. Providing a detailed overview of how data, from both the clients, and database are transmitted. Showing how the user interface has changed from the mock ups, and how the user interface reacts to user's actions. Chapter 6 will be providing an overview of how well the project progressed, and how the students dealt with using the cutting edge technology.

2 FEASIBILITY & REQUIREMENTS

2.1 REQUIREMENTS ANALYSIS

2.1.1 User Requirements

When entering the website for the first time, the user will be able to register an account, or log in using an existing account.

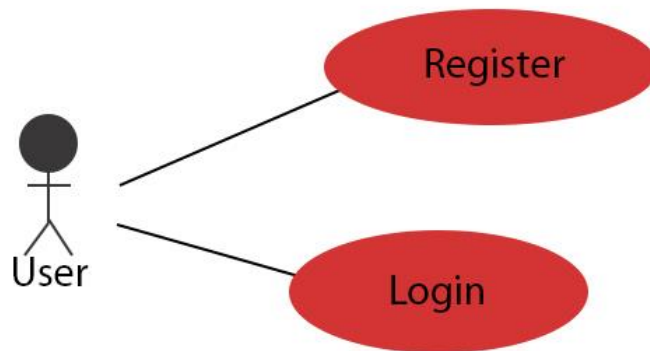


Figure 3.1 - User Registration

Once logged in, the user can do a number of tasks. Users can add other users as friends remove friends and view friend's details.

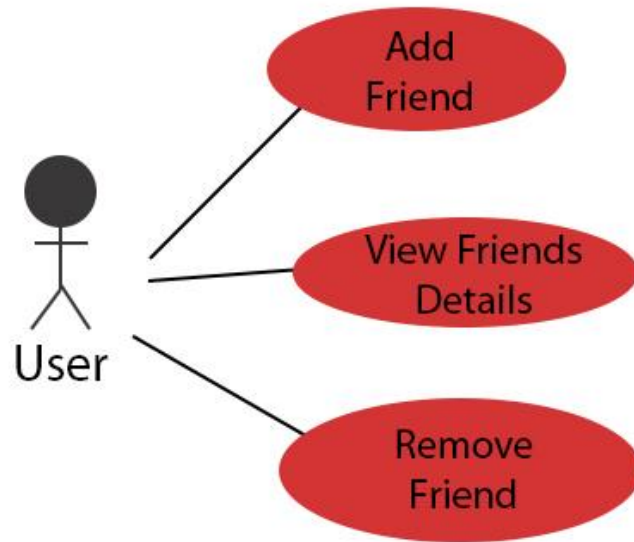


Figure 3.2 - User Friends

Users can call any user on their friends list and answer any incoming calls from friends. Users can send/receive messages with their friends.

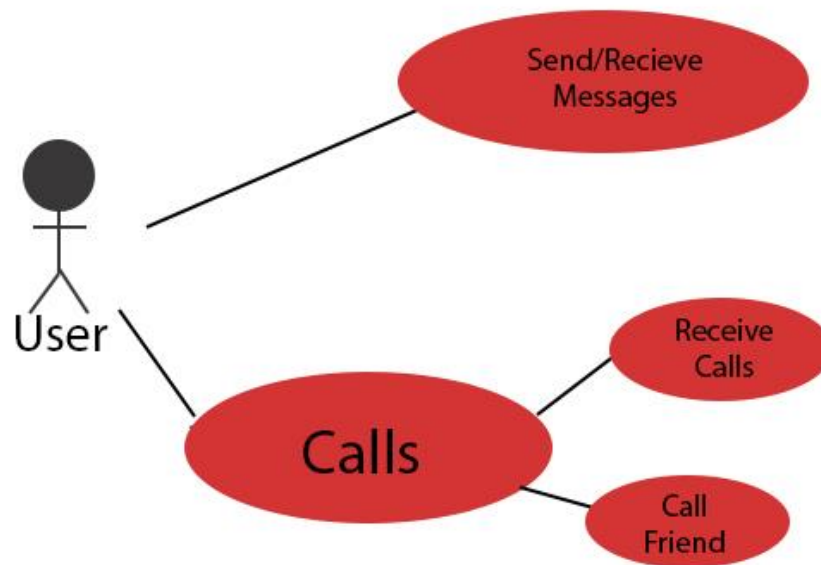


Figure 3.3 - User Calling

2.1.2 Functional Requirements

When the user enters the website they are presented with two options, to Login using an existing account or to register a new account. Registering requires the user to enter an email, a password and a confirmation for the password. Once registered, the user can sign in with their email address and password.

Once logged in the user is brought to their home page. This page displays all their friends. Each of their friend's details can be viewed. From this page friends can be added or removed. Once a user is added as a friend, the user can send and receive messages and calls with the other user. The home page will look like this.

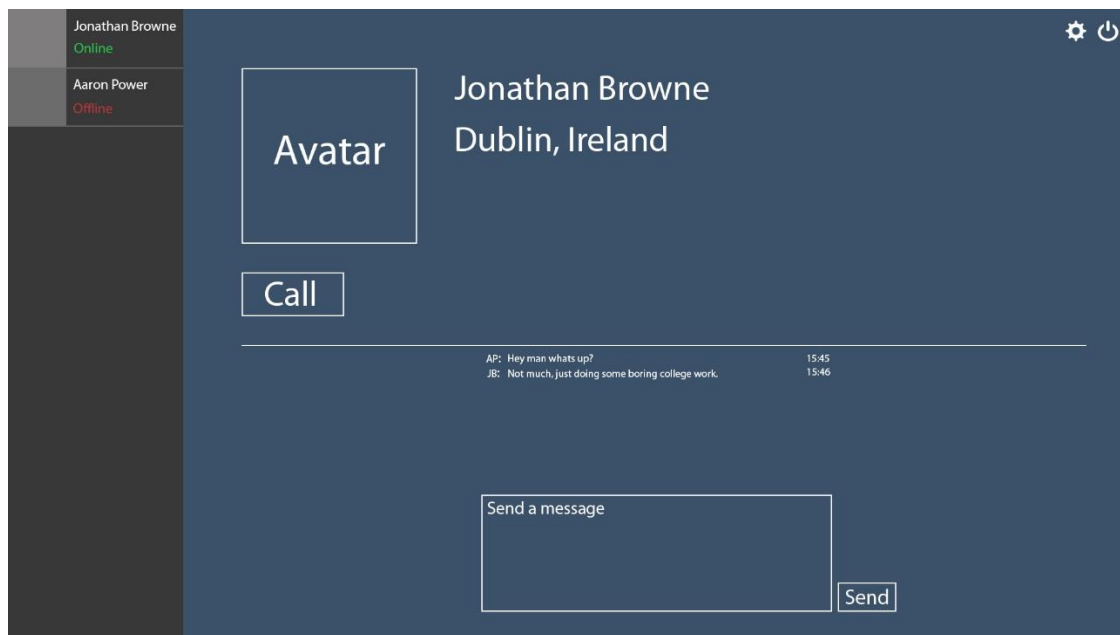


Figure 3.4 - Home Screen Mockup

All the users' friends will be displayed on the left side. The main portion of the screen will show their friends info and any text conversations. This section will also allow the user to call their friend.

Across the top of the page is the logout button, and links to edit their account settings.

2.1.3 Non-Functional Requirements

The non-functional requirements for this project would include a number of security measures to protect the information of the users using the website.

2.1.4 Technical Requirements

A number of different technologies will need to be learned.

Mongo DB: A no SQL database. This will be the database for the application. Mongo is document oriented and stores data in JSON-like objects instead of tables.

Node.js: A JavaScript based runtime tool for sever side code. This in conjunction with Mongo allows the server side code to be written entirely in JavaScript.

Sass: This is a pre-processor extension for CSS3. Sass adds a number of features like variables and inheritance thus making CSS more powerful.

Jade: A template language for HTML uses a number of shorthand's for HTML allowing coding to be done quicker.

Angular JS: A JavaScript framework allowing the creation of powerful single page web applications.

GULP: An automation build system based on streams. The configuration systems is based on JavaScript.

Socket.IO: A JavaScript library for real-time web applications. It Enables communications between web clients and the server. It works in conjunction with node.js on the server side.

2.2 SYSTEM MODEL

The system is divided between 3 components. The Database (MongoDB), the Server (Node.JS, Socket.IO), the Client (Angular.JS). The client would take in the user's input (The data stream taken from the user's microphone, the user's login information), and then pass that information through a RESTful API, or over WebSockets.

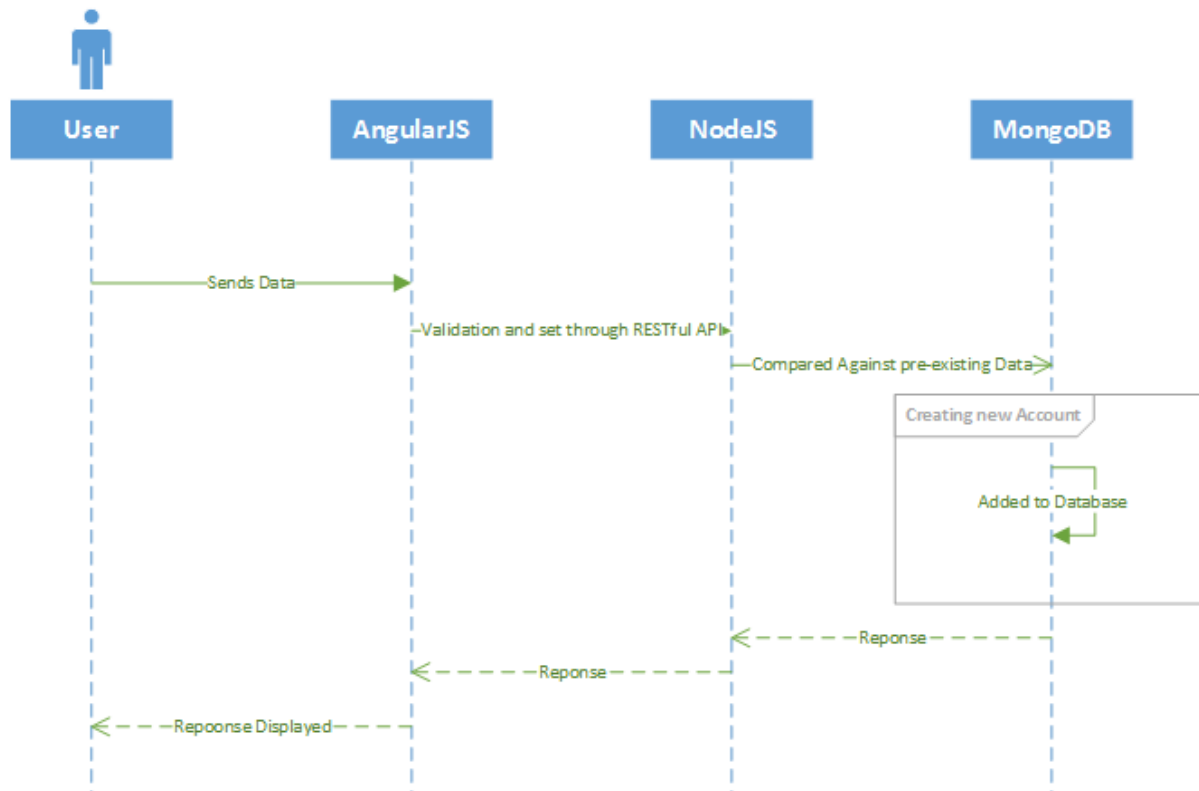


Figure 3.5 – System Overview

REST (Representational state transfer) is set of standards that set how concepts such as HTTP and URI's are used. A RESTful API is applying these standards to your own application for ease with the transfer of data between your client and your server.

This data is then processed and if relevant, it will either pass information to another client as requested or the data is logged into or checked with validation against the data in the database.

2.3 FEASIBILITY

There are many problems with implementing a live web audio system. Most of the problems occur with NAT(Network Address Translation). Most of these problems can be fixed with enough manual testing.

The project management risks stem from is that only two people are developing this project and thus very dependent on one another to make sure the sides of the project

work in tandem with each other and if one of us is not able to finish the work in time it would hinder the entire project.

This will be solved by additional meetings to the supervisor meeting to make sure that the project stays on track. The project will also be using GitHub to ensure that the code being run is latest code developed.

This solves the problem of the project developers being both located quite a distance from each other and just manually sending over files every change over emails and such would be tedious.

3 RESEARCH & ANALYSIS

3.1 SYSTEMS ANALYSIS

3.1.1 User Registration/Sign in

When the user enters the website they are presented with the option to sign in or register. When choosing to login in, the user must enter their username and password into a form. The data from the inputs will be validated using JavaScript. Their details will then be compared to information in the database, if their information matches that in the database then the user is signed in. Underneath the input fields there is the option to register. Clicking this link will open another form, here the users are required to create a username and password. The username entered is validated against the database to ensure that two users do not have the same usernames. The users password is validated with a confirm password field using JavaScript.

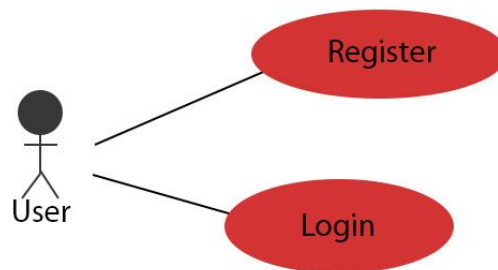


Figure 4.1 - User Registration/Login

3.1.2 Add/View/Remove Friends

Once signed into the website the user is brought to their home screen. On the home screen the user's friends list is displayed on the left side of the screen. To the right is the information of the friend that is currently selected. The website will be made using AngularJS, a JavaScript library that is used to create a single page application (SPA). The home screen will use AngularJS so that when a friend from the list is selected the data on the right side of the screen will update without having to reload. To add a friend, the user can enter the required user's username into a search bar, the database will then be queried and any related responses will be returned. Once a desired user is found, a friend request can be sent. Once sent the user receiving the request has the option to accept or refuse the friend request. Once the user is added, that friend is added to an array of users associated with the user. For friend deletion, the user has the option to delete a friend from their friends list in that specific user's information. The database then removes the friend from that users list of friends.

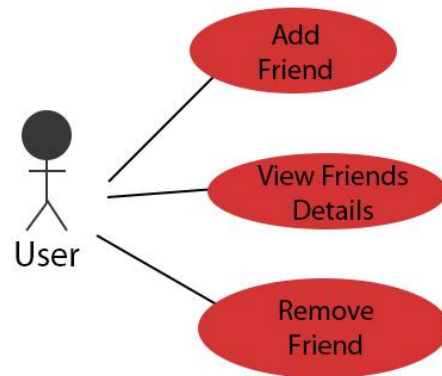


Figure 4.2 - User Friend Options

3.1.3 Call/Message Users

Once a user is added to another user's friends list, that user can now call and message that user. To call a user, first the user is selected from the friends list. In the users page there is a call button, once the button is pressed a call request is sent to the other user, the other user then has the option to accept or refuse the call. Once the call is accepted, the user's web cameras and microphones data is retrieved using the getUserMedia JavaScript API. The data from each camera/microphone is sent via web sockets (Socket.io). In the call, each user has the option to change the volume, hang up the call and mute their microphone. A log of all text messages sent between both users is kept.

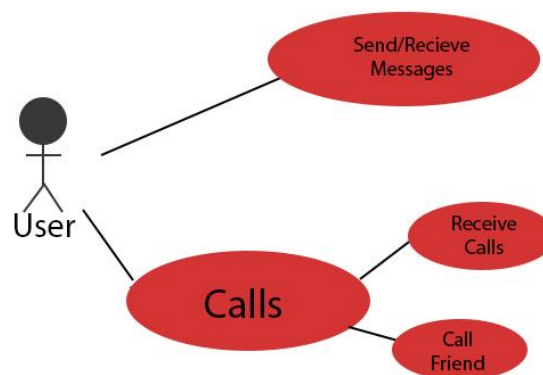


Figure 4.3 - User Calling Options

3.1.4 User Interface

3.1.4.1 Splash Screen



Figure 4.4 - Splash Screen Mockup

This is the first page the user sees when entering the website. The user is given two options, to register as a new user or to login to an existing account. Upon clicking either link brings the user to the relevant form.

3.1.4.2 Register Screen



Figure 4.5 - Register Screen Mockup

Here the user is required to give some basic information to create an account. If there is an error on the form, then a red indicator appears on the relevant input field.

3.1.4.3 Login Screen

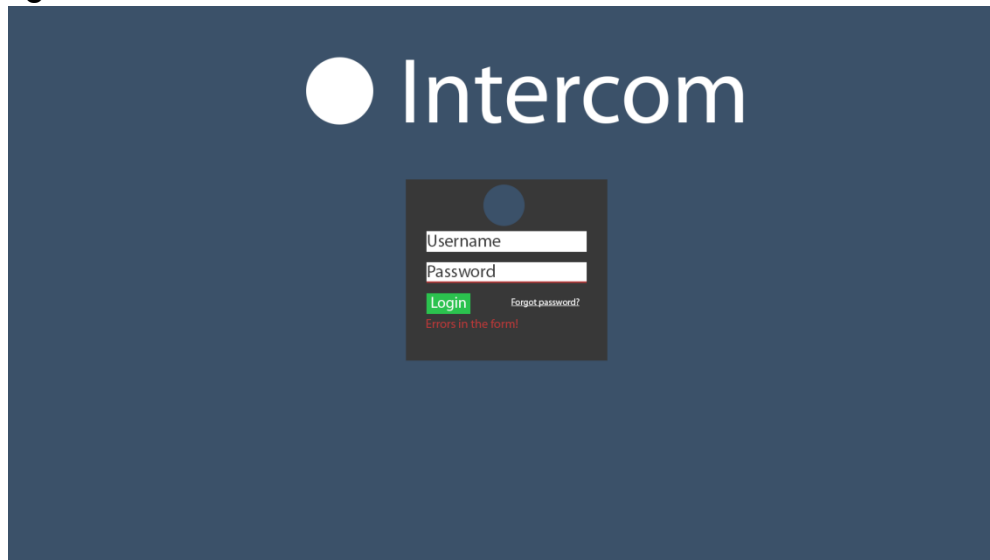


Figure 4.6 - Login Screen Mockup

Here the user is prompted to enter their login details. The form is validated using JavaScript, like the registration form, an error message is displayed if there is a problem validating the form. A red indicator highlights the input field that is causing the error.

3.1.4.4 Home Screen

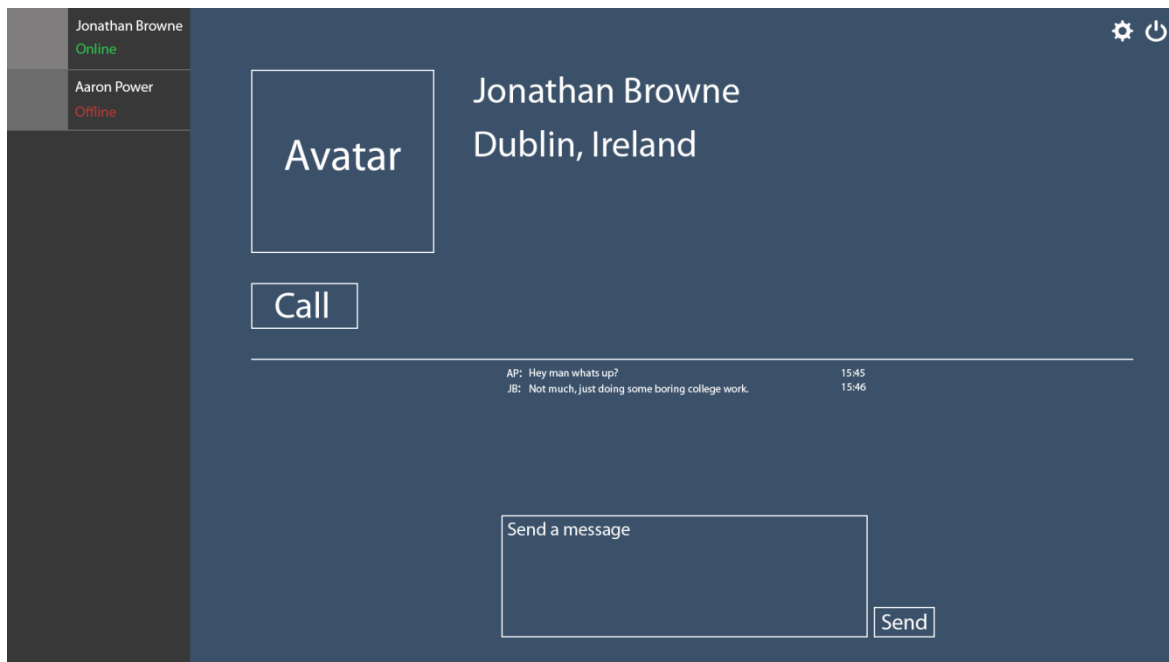


Figure 4.7 - Home Screen Mockup

This is the main page of the website when the user is signed in. On the left side of the screen is the user's friends list. The friend's username, avatar and current online status is visible. Once a friend is selected, the right side of the screen displays that friend's information. Using AngularJS, when a friend is selected the page does not reload, the page is updated. The friend's information contains their name and location and their avatar/profile picture. Here the user is able to call the friend using the call button. At the bottom of the page is a log of all the text messages sent between the two users. This log has record of who sent what message and a time stamp of when the message was sent.

3.2 IMPLEMENTATION PLATFORM REQUIREMENTS

3.2.1 JavaScript

In order to fulfil the technical requirements the students will be required to build on their previous experience in JavaScript. While using a skill previously taught, the JavaScript that will be written will require a high understanding, and ability to utilize JavaScript, and ECMAScript[1] (A standard set by ECMA International on client side scripting on the web. Implemented in most browsers [2]).

In this project JavaScript will not just be used for client side scripting, but rather it will encompass the entire scope. From build tools, to server side logic, to database storage.

3.2.2 Node.JS

Node.JS is an asynchronous event driven platform built on Google's V8 JavaScript engine originally created in 2009 by Ryan Dahl and now run by Joyent[3][4]. Node.JS has become hugely popular with Companies like Microsoft, Yahoo!, LinkedIn, and PayPal all use Node.JS for handling parts of their business [5].

The project required a fast back-end to support the project's constant signaling between clients, and reading or writing to the database all at the same time without slowing down the user's experience. This made Node.JS a perfect fit. In addition to its fast event based I/O (Input/Output) model, it allows the project's backend to be written in JavaScript maintaining consistency across the front and back end.

The project will be using Node.JS for serving our webpages, signaling between clients for WebRTC (Explained in WebRTC), and accessing our database and passing the data to the client.

3.2.3 Express

Node.JS isn't explicitly designed for being a HTTP server. Designing and implementing a http server from plain node to the standard required by this project, would require too

much time, not even taking into account the time to optimize it to not only be capable, but be fast.

For this reason the project will be using Express. Express is a web framework sponsored by Strong Loop [6]. Express will allow the project to easily serve webpages and create a REST (Representational state transfer) API for delivering data to the client.

3.2.4 MongoDB

For this project to achieve the speed and reliability required the project will be using a NoSQL database. Specifically we will be using a document-oriented database called MongoDB. MongoDB released in 2009, is the current leading NoSQL database [7].

We will be using 'mongoose' a node module for interacting between node and MongoDB. The data will be transferred into JSON (JavaScript Object Notation) objects and stored into the database. Its use of JSON objects is also important as to keep consistency between data types.

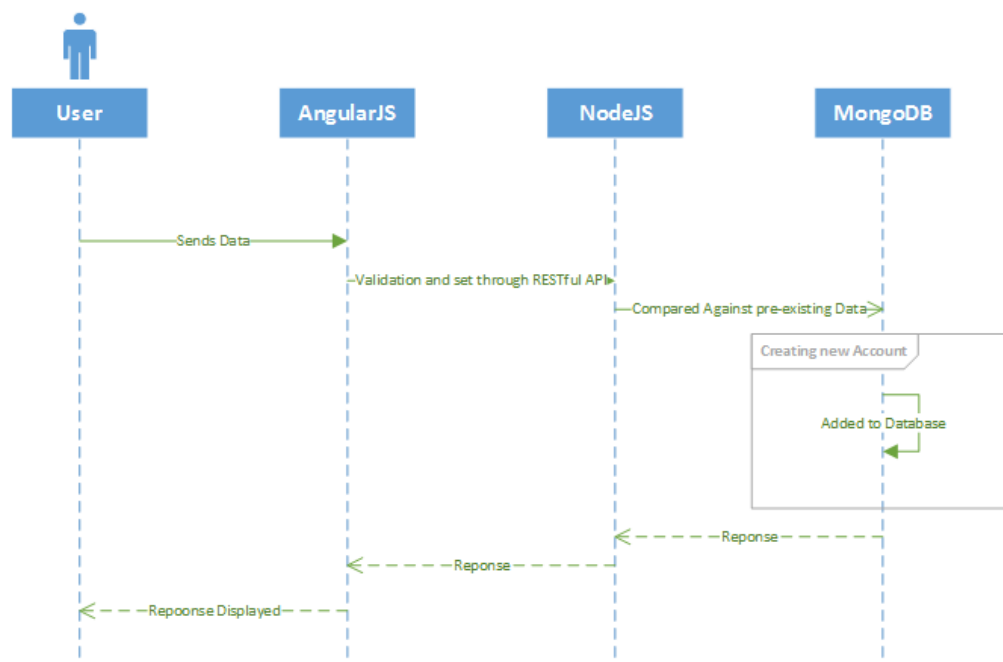


Figure 4.8 - User Registration

3.2.5 AngularJS

AngularJS is a client side framework Sponsored by Google, released in 2009. AngularJS is designed for creating dynamic single page applications. Just this reason alone makes it perfect for this project, a single page web communication app.

Its main use in this project is to design the project modularly, and make the behavior contained within the JavaScript files are clear, and to allow the project to manipulate the DOM (Document Object Model) and link it with our back-end with ease. One

example of this will be a user's friends list which will use an angular template to dynamically create a list from that data onto the DOM.

3.2.6 Gulp

While Gulp is not a visible to the user that the project uses Gulp, it is one of the most essential technologies used in this project. Gulp is a build tool which uses node's stream objects to pass data. A Stream is an abstraction interface similar to pipelines in Unix-like systems [8]. Streams are very flexible, they can be readable, or writeable, or both (Duplex) [9]. Gulp is designed to use streams to take the data from the projects source files and interact with them in some way. The project also uses plugins on top of vanilla Gulp to perform certain actions, like performing operations on AngularJS code.

For example the current Gulp build will clear the previous builds files, then take the client side JavaScript, lint against JSHint (A JavaScript Linter) it will stop there if the code has errors, it will also do this again if the code has been changed since it last ran. It will then minify (The process of condensing code to the smallest possible size), and also insert a source map for development (A source map allows the students to debug minified code).

With AngularJS code, Gulp will do some operations to allow the projects code to maintain and even improve page load times. Gulp will also perform similar operations on the server code. Gulp has also been programmed to start up the Database allowing the students to start developing and testing with just the use of Gulp's CLI.

3.2.7 WebRTC

WebRTC (Web Real Time Communications) is an open source framework/API by Google, Mozilla, and Opera for delivering video and audio across peers [10]. WebRTC is divided into three main categories RTCPeerConnection, RTCDataChannel, GetUserMedia [11].

RTCPeerConnection is for streaming data between peers. This eases the load placed on our servers offloading it for peers, this wouldn't be ideal in a use case where there are many peers all connected to each other as it would cause significant performance on the host of the peer connections. This would be solved with a Media Server that would initiate when there are a significant amount of users sharing connections, however that maybe out of the scope of this project [12].

GetUserMedia allows us to access the user's microphone, and/or camera (with their permission). This along with RTCPeerConnection make up the core of our functionality [13].

RTCDataChannel is for sending arbitrary data through a peer connection that would not need to be continuous. For example calling a user would require continuous back and forth and is thus suited to RTCPeerConnection, while sending a file to a user is one

way and ends once the file has been fully delivered. RTCDataChannel will probably go unused for this project as it is outside the current scope. It could however be added to the project if the project's goals are met sooner than anticipated. Adding file sharing functionality [14].

For this project rather than access the WebRTC API directly, this project will use a library named WebRTC.io.js. Currently most WebRTC libraries are actually API where the owner's host the servers. This wouldn't be ideal for our requirements, as we want to handle the data and how it is sent. WebRTC.io currently only abstracts how to handle the data but does not subvert us handling the data. For example a Firefox's Hello application (A competitor to our project) uses the OpenTok library by TokBox [15]. To use this library for the project it would be required to pay \$50 per month plus an additional amount per number of minutes after 10,000 minutes worth of calls have been used on our servers [16].

However the problem that arises with this library in terms of the code is that it has not been updated since March 6th 2013[17]. This could cause issues, however if necessary the students are willing to either fork (Github term to mean take the existing code and modify) the library, or if the problem is pervasive enough throughout the library look for another.

3.2.8 Socket.IO

Socket.io uses WebSockets to allow the project to send data between users in real in events. Socket.IO will be used for the chat functionality as unlike video/audio calls a constant connection is not needed. Data only needs to be sent when the user sends a message. Socket.IO would also be used for a media server as mentioned earlier if the project is finished in time.

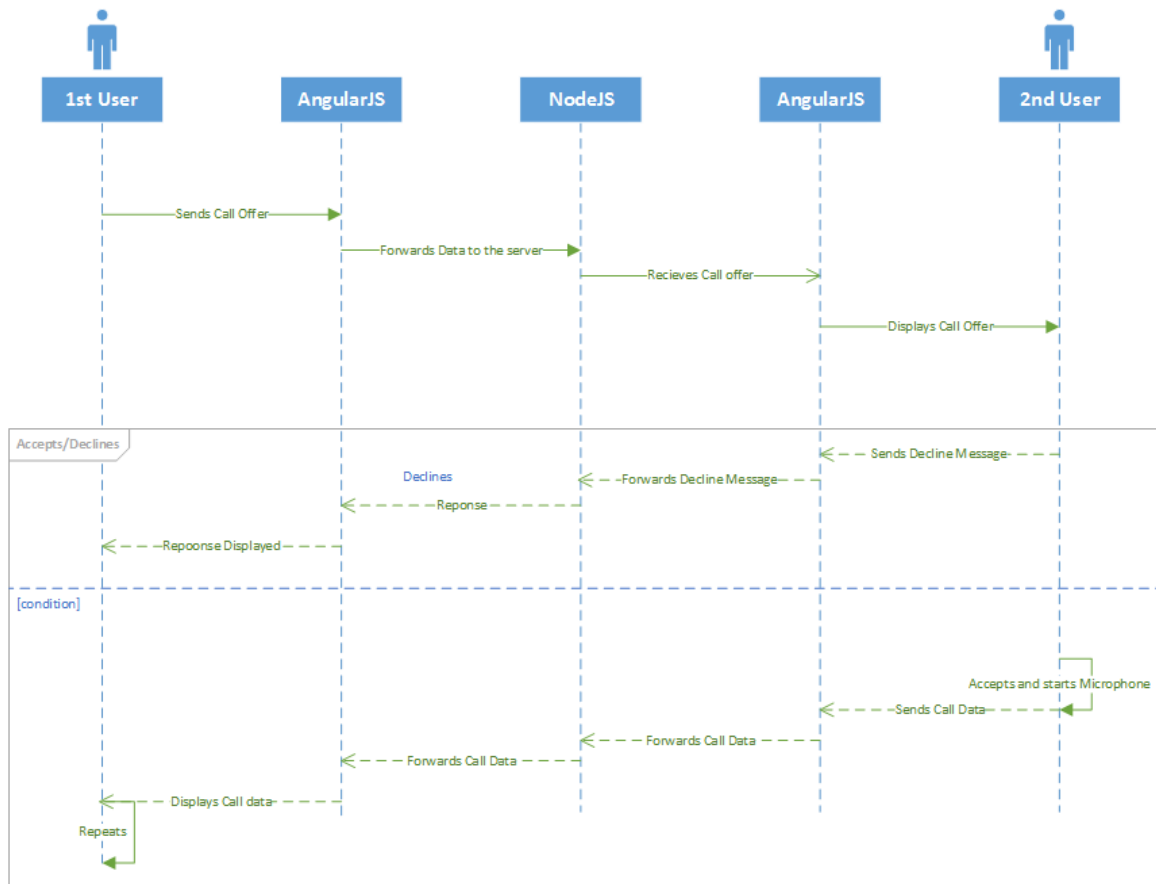


Figure 4.9 - User Calling

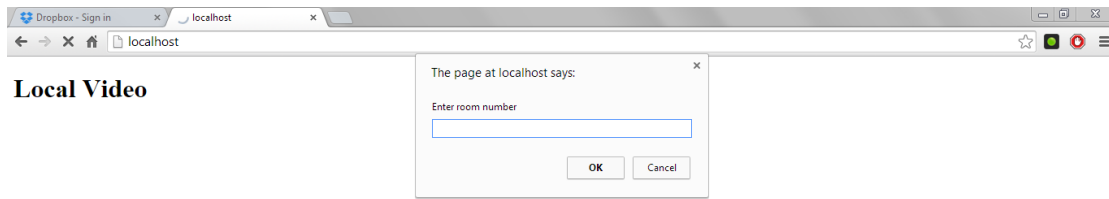
3.2.9 Jade & Sass

To make the process of generating, and displaying the projects webpages easier for the students they have opted to use Jade, a HTML templating language, and Sass (Syntactically Awesome Stylesheets) a CSS (Cascading Stylesheets) pre-processor.

Jade is a HTML templating language that is coupled with Express. It is designed to allow templating and insertion of data into the file before being rendered into a HTML file for the user. While Jade does not provide a faster delivery of content, it is designed to allow the students to have an easier time writing code, and reduces code re use.

Sass is a CSS pre-processor designed by Hampton Caitlin. Which means that it compiled into CSS before being uploaded to the server. The user nor the server it is hosted on know that the CSS file was created with Sass. It is however extremely useful for writing good CSS. Allowing Mixin's and variables significantly reducing code re-use. Sass like Jade is more of a quality of life improvement for the students over their users.

3.3 PROTOTYPE

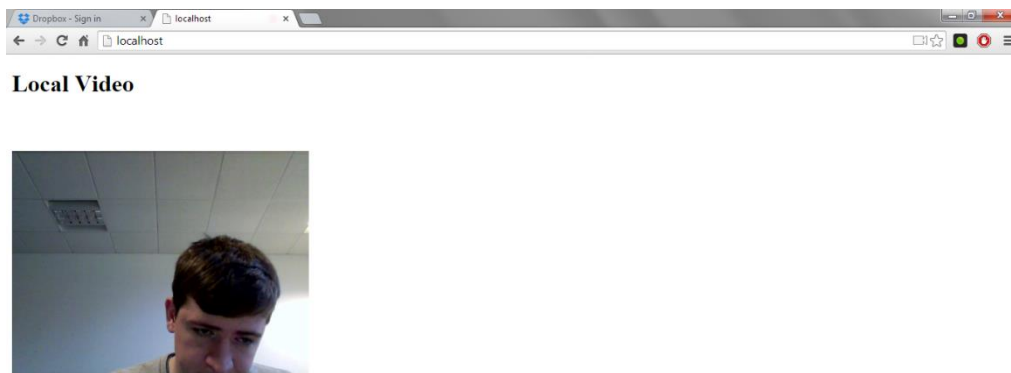


Remote Videos



Figure 4.10 - Screen 1

User one enters a room. In future the rooms will be not entered, but generated from a random string to ensure users don't enter the wrong call.

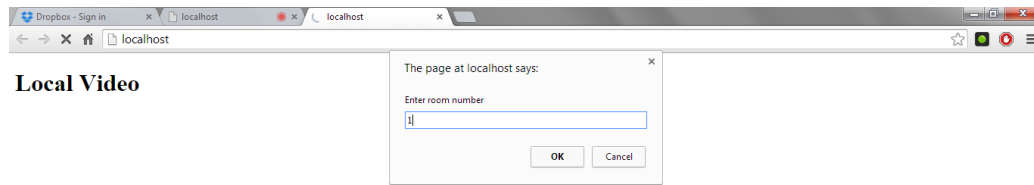


Remote Videos



Figure 4.11 - Screen 2

User one's local video is displayed



Remote Videos



Figure 4.12 - Screen 3

User 2 connects to the same room.

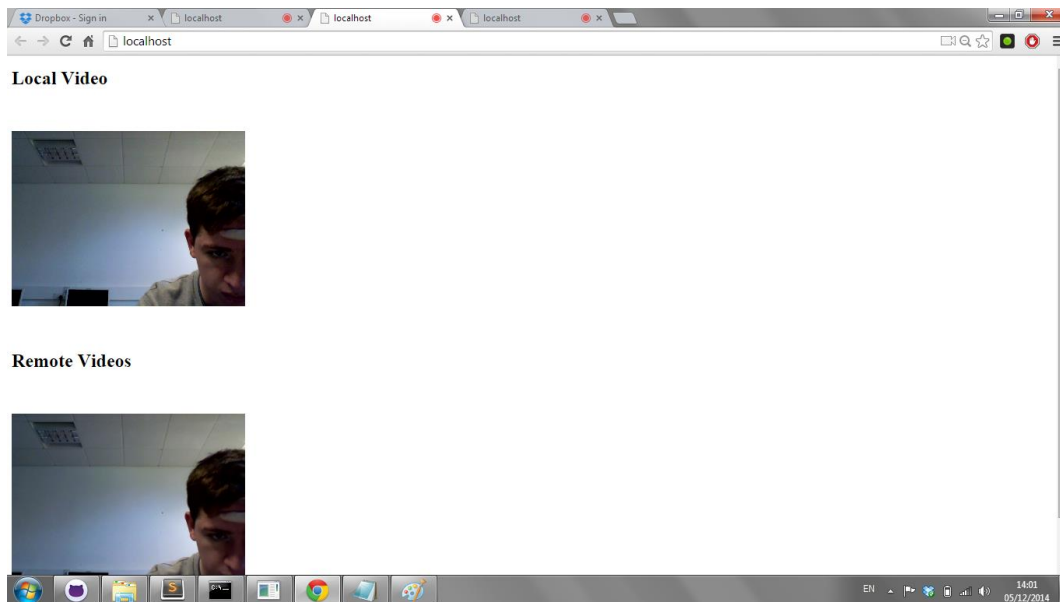


Figure 4.13 - Screen 4

The one to one call initiates.

4 DESIGN

4.1 CLIENT SIDE

4.1.1 Registration Screen



Figure 5.1 - Register Screen Second Version

Registration for the app requires the user to enter a username, password and then confirm the password for validation. The forms for the site are validated using AngularJS modules.

```
1. <form name="form">
2.   <ul>
3.     <li>
4.       <input type="email" placeholder="email" name="email" ng-
      model="user.email" required />
5.       <span ng-show="form.email.$dirty && form.email.$error.required">Required!</span>
6.       <span ng-
      show="form.email.$dirty && form.email.$error.email">Not a valid email!</span>
7.     </li>
8.     <li>
9.       <input type="password" placeholder="password" name="password" ng-
      model="user.password" required ng-minlength="5" ng-maxlength="10" />
10.      <span ng-
      show="form.password.$dirty && form.password.$error.required">Required!</span>
11.      <span ng-
      show="form.password.$dirty && form.password.$error.minlength">Password too short!</span>
```

```
12.         <span ng-  
show="form.password.$dirty && form.password.$error.maxlength">Password too long!</span>  
13.         </li>  
14.     </li>  
15.         <input type="password" placeholder="confirm password" name="passwordConfirm" ng-  
model="user.passwordConfirm" required ng-minlength="5"  
16.             ng-maxlength="10" />  
17.         <span ng-  
show="form.passwordConfirm.$dirty && form.passwordConfirm.$error.required">Required!</span>  
18.         <span ng-  
show="form.passwordConfirm.$dirty && form.passwordConfirm.$error.minlength">Password too shor  
t!</span>  
19.         <span ng-  
show="form.passwordConfirm.$dirty && form.passwordConfirm.$error.maxlength">Password too lon  
g!</span>  
20.         </li>  
21.     </li>  
22.         <button class="button-primary" ng-  
disabled="form.$invalid && {{ user.password !== user.passwordConfirm }}">Register</button>  
23.     </li>  
24. </ul>  
25. </form>
```

A number of modules control what the user is required to enter into the different inputs. For example a user needs to enter a valid email address using {a-z, 0-9}@{a-z, 0-9}.{a-z} if the data is different than what is required, a span appears informing the user of the error and the input box where the data is incorrect is highlighted in red. The submit button at the bottom of the form is disabled if there are any errors, this is what allows the data to be valid upon submission as it is impossible for the user to enter the form unless everything is correct.

4.1.2 Login Screen

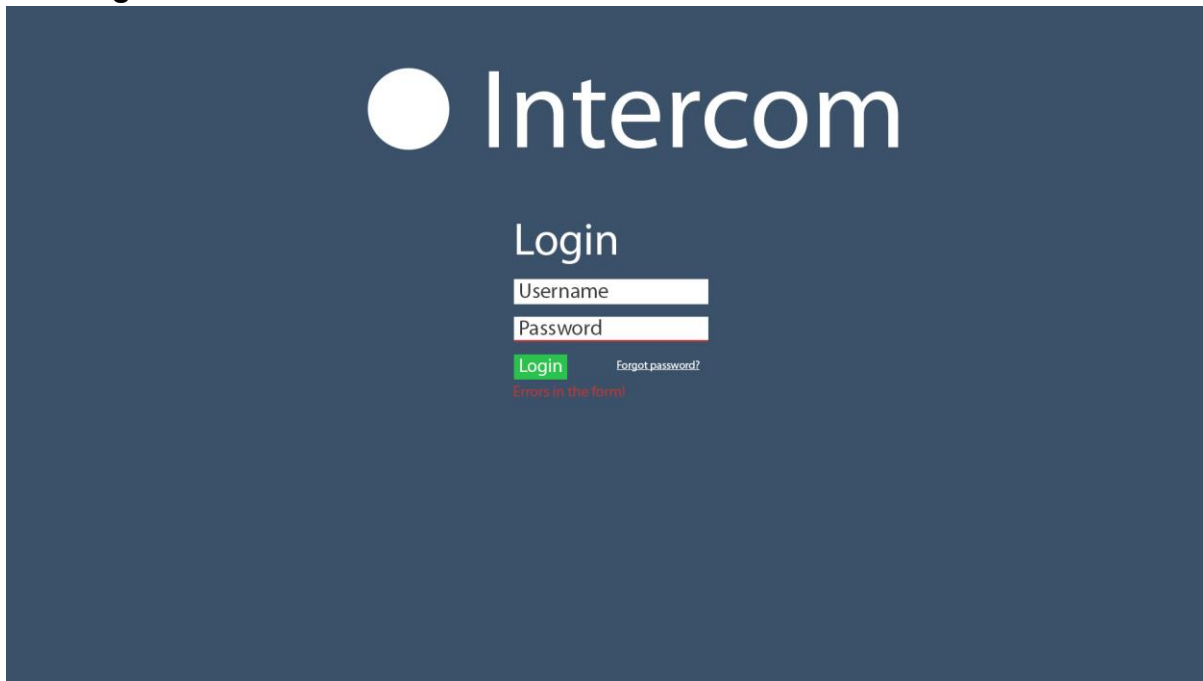


Figure 5.2 - Login Screen Second Version

The login process functions the same as the registration process. Users are required to sign in using their username and password. The form is validated using AngularJS. This form functions the same as in if the data is incorrect then the submit button is disabled and a span informs the user what input field is incorrect.

4.1.3 Main Screen

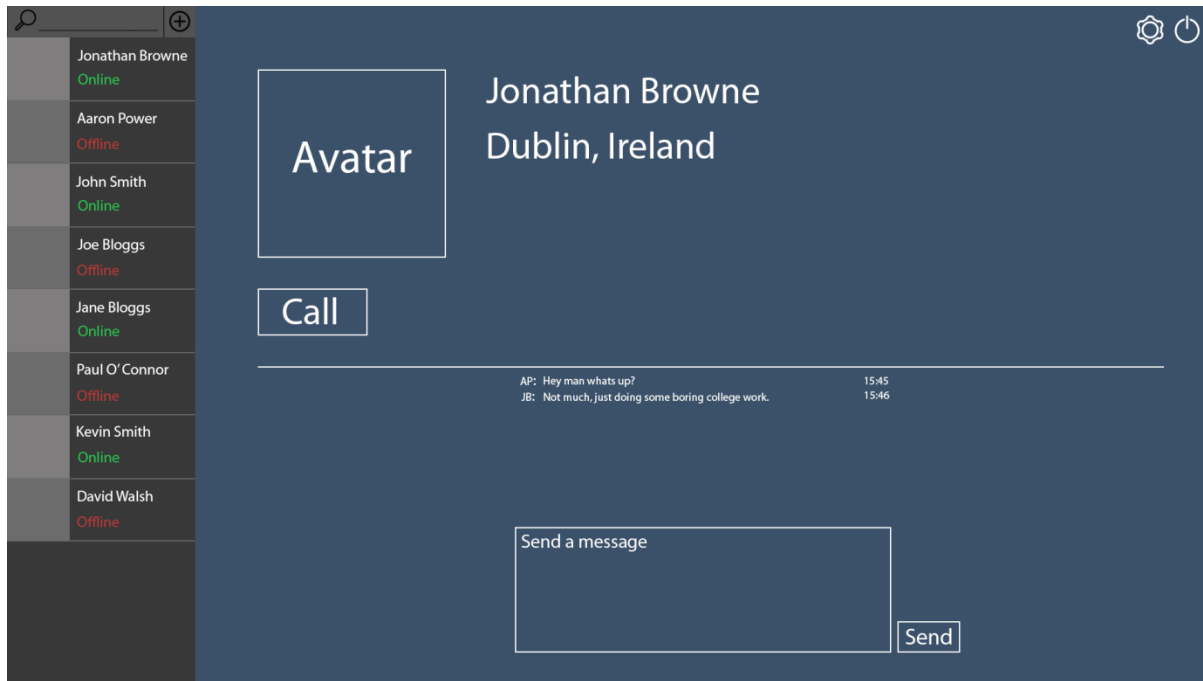


Figure 5.3 - Main Screen Second Version

Once the user signs in, they are sent to the main screen of the app. From here the user can call other users and message them. Users can only call and message users if they are added to their friends list. At the top of the friends list is a search bar for filtering through the user's friends list. There is also a add button that when pressed brings up a form for adding a user to their friends list. Once a user is added if their name on the left is clicked it brings the user to that users details screen. From there the user can call their friend or message them. The message log contains all their previous messages to one another. In the top right is the logout button for the app and a settings button that will display a settings menu for the user to change account details such as their password and profile picture. The change between friend's details screens is dynamic and does not require the page to reload. This is done using AngularJS.

4.1.4 Call Screen

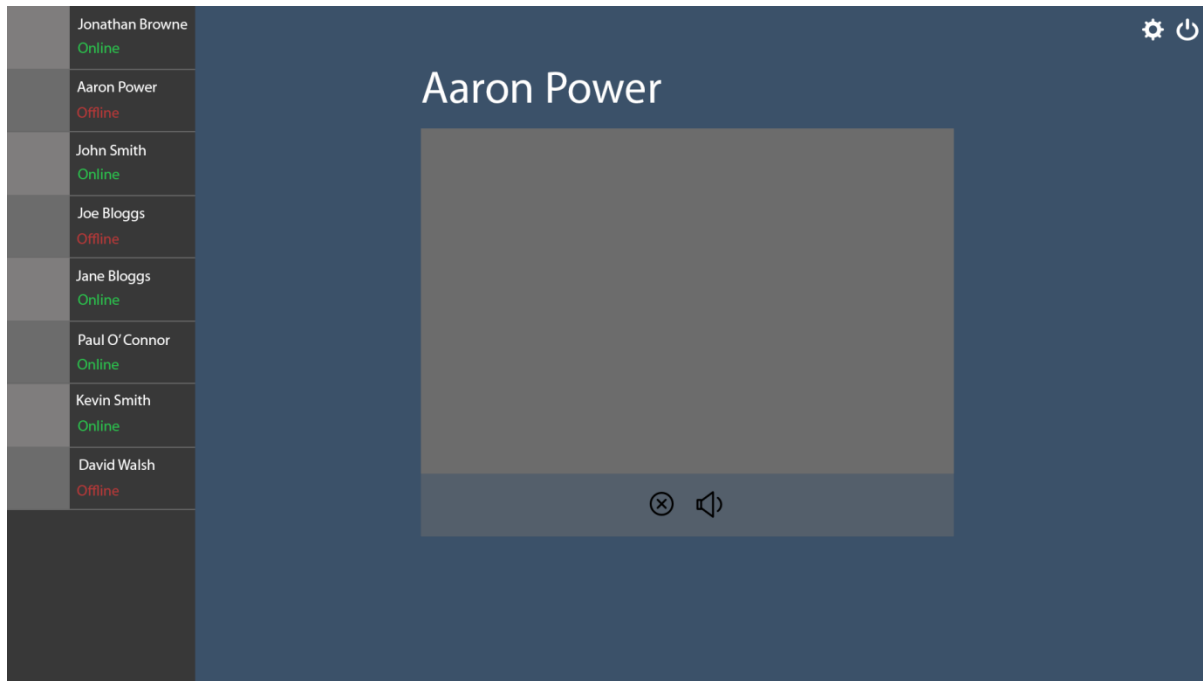


Figure 5.4 - Call Screen Mockup

Once the user presses the call button for a friend on their friends list they are brought to the call screen. The call will then be connected to the friend, once the call is up and running the user has the option to mute the volume of the call or to press the end call button. Once the end call button is pressed the user is brought back to the details screen of that particular friend. The user's friends list is still displayed on the left side of the screen during the call to inform users of any other friend that may come online.

4.2 SERVER SIDE

The server side of the project features a fully functional REST API for accessing, and serving the database to the client. The server side also renders the jade templates into HTML. The server also utilizes Socket.IO for real-time messaging and storing within the database.

4.2.1 NodeJS

In order to make use of Node's asynchronous capabilities the project must use Node's callback functionality. As shown below.


```
1. object.prototype.method = function (callback) {  
2.   var data  
3.     , error  
4.  
5.   // create data, and error if there is a problem  
6.  
7.   callback(error, data)  
8. }
```

```
1. object.method(function (error, data) {  
2.   if (error)  
3.     // Handle error if it exists  
4.  
5.   // Make use of the data retrieved.  
6. })
```

This is required in order to make use of node modules.

4.2.2 Express

We're using Express, a HTTP framework for node to handle our web server. The servers' main purpose is to provide a REST API for the client to access the data needed.

4.2.3 REST API

A REST API allows for the client to send information to the server without having to worry how the information is handled. This is especially useful for using a database as having the client making direct calls to the database could be vulnerable to malicious attacks. Abstracting the information behind HTTP requests allows the server to be in control of how the information is delivered and handled. The client then only has to worry about whether they got the information they wanted or they failed.

Some requests require authorization, which is done by having a JSON web token in the authorization http header, we then check the token against the tokens stored in our database. If the token matches the user is attached to the request object. If the token doesn't match any tokens located in the database a "401: Unauthorized" error is sent.

Unless specified the success is sent in JSON format.

User - Request the users' friend's.

0.0.0 ▾

GET

localhost/api/users/

Header

Field	Type	Description
Token	String	Access key.

Success 200

Field	Type	Description
List	Object[]	of friends.

Success Response:

```
HTTP/1.1 200 OK
[
  {
    "_id": "54c9050cbd7420e0074fb90c",
    "avatar": "smileyface.png",
    "username": "",
    "email": "johndoe@email.net"
  },
  {
    "_id": "54e35727a8d330d81d001ab8",
    "avatar": "",
    "username": "Jane Doe",
    "email": ""
  },
  {
    "_id": "54e479e31ac5c5702aa797d2",
    "avatar": "",
    "username": "",
    "email": "johnsmith@test.com"
  }
]
```

Error 4xx

Field	Description
403	No Token was placed into the request
404	No User was Found with that token

404 Response: 403 Response:

```
HTTP/1.1 404 Not Found
```

Figure 5.5 - Get Users Friends

User - Signin a user.

0.0.0 ▾

POST

localhost/api/users/signin

Success 200

Field	Type	Description
Token	Object	Users' token

Success Response:

```
HTTP/1.1 200 OK
{
  "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJhdmF0eXkiOiIiLCJ1c2VybmFtZSI6IiIsImVtYWlsIjoiaWVtZWVzZGVyM0B0ZXN0L"
}
```

Error 4xx

Field	Description
404	No User was Found with that token

404 Response:

```
HTTP/1.1 404 Not Found
```

Figure 5.6 - User Sign in

User - Save a new user.

0.0.0 ▾

POST

localhost/api/users/

Body

Field	Type	Description
email	String	Users' email
password	String	Users' password
confirm	String	Users' confirm
username	String	Users' username
avatar	optional String	Users' avatar

Success 200

Field	Type	Description
Token	Object	Users' token

Success Response:

```
HTTP/1.1 200 OK
{
  "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJhdmF0eXkiOiIiLCJ1c2VybmFtZSI6IiIsImVtYWlsIjoIdGVzdGVyM0B0ZXN0L"
}
```

Error 4xx

Field	Description
403	Details are incorrect.
409	A user with that email already exists.

403 Response: 409 Response:

```
HTTP/1.1 403 Forbidden
```

Figure 5.7- User Account Creation

User - Update a Users' information.

0.0.0 ▾

PUT

localhost/api/users/

Body

Field	Type	Description
email <small>optional</small>	String	Users' email
password <small>optional</small>	String	Users' password
confirm <small>optional</small>	String	Users' confirm
username <small>optional</small>	String	Users' username
avatar <small>optional</small>	String	Users' avatar

Success 200

Field	Type	Description
Token	Object	Users' token

Success Response:

```
HTTP/1.1 200 OK
{
  "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJhdmF0eXkiOiJLCj1c2VybmFtZSI6IiIsImVtYWlsIjoiaGVzdGVyM0B0ZXN0L"
}
```

Error 4xx

Field	Description
403	No Token was placed into the request
404	No User was Found with that token

404 Response: 403 Response:

```
HTTP/1.1 404 Not Found
```

Figure 5.8 - Update User Details

User - Accept a users' friend request.

0.0.0 ▾

POST

localhost/api/users/accept

Success 200

Field	Type	Description
Token	Object	Users' token

Success Response:

```
HTTP/1.1 200 OK
{
  "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJhbmF0eXkiOiIiLCJ1c2VybmFtZSI6IiIsImVtYWlsIjoiaGVzZGVyM0B0ZXN0L"
}
```

Error 4xx

Field	Description
403	No Token was placed into the request
404	No User was Found with that token

404 Response: [403 Response:](#)

```
HTTP/1.1 404 Not Found
```

Figure 5.9 - Accept Friend Request

User - Remove a user.

0.0.0 ▾

POST

localhost/api/users/remove

Success 200

Field	Type	Description
Token	Object	Users' token

Success Response:

```
HTTP/1.1 200 OK
{
  "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJhdmF0eXkiOiIiLCJlc2VybmFtZSI6ImVtYyIjoiidGVzZGVyM0B0ZXN0L"
}
```

Error 4xx

Field	Description
400	User isn't a friend
403	No Token was placed into the request
404	No User was Found with that token

400 Response: 404 Response: 403 Response:

```
HTTP/1.1 400 Bad Request
```

Figure 5.10 - Remove User from Friends List

0.0.0 ▼

```
localhost/api/users/validate
```

Field	Type	Description
Token	String	Access key.

Field	Type	Description
Users	Object	' token.

Field	Description
403	No Token was placed into the request
404	No User was Found with that token

HTTP/1.1 404 Not Found

4.2.4 MongoDB

40

Entity Relationship diagram:

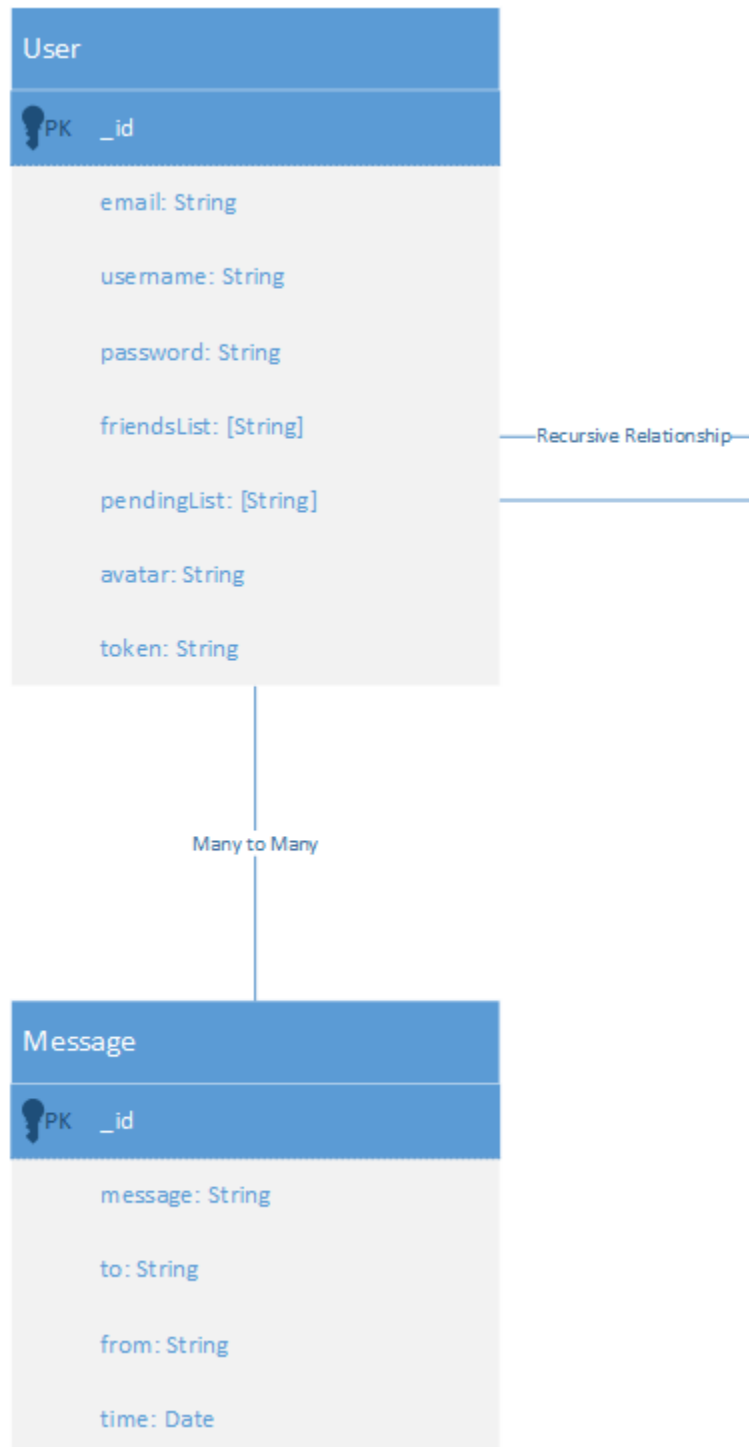


Figure 5.12 - Database ERD

5 IMPLEMENTATION

The code-base was stored on GitHub [18]. Using Git [19] which is a distributed version control system, the students could access, edit and publish the changes without worry of code being lost or unintentionally overwritten. GitHub also keeps track which student written which parts of the code-base. Commits marked as unknown should be attributed to both students as the commits were from a college computer [20].

5.1 TECHNOLOGY/LIBRARY SUMMARY

Node.JS: A Runtime environment built on Google's V8 [21] JavaScript engine.

NPM [22]: A Package Manager for Node.JS

Bower [23]: A Package Manager for client-side libraries.

Gulp [24]: A Build, and automation system built on Node.JS.

5.1.1 NPM

All Node modules that are not part of the standard library [25] are stored on NPM. NPM is included by default with Node.JS. To install a node plugin run the following command in your project directory from the command line.

```
C:/User/.../Echo> npm install moduleName
```

Annotation: Figure I: Installing a node module

```
1. var module = require('moduleName')
```

Annotation: Figure II: The require syntax.

This will allow use of the module from within a file using the require syntax as shown in figure II. This will return a Function object that can then be used. Some modules need to be run directly from the command line. This is achieved by adding '-g' or '--global' flag to the command shown in Figure I. These modules are then stored in a sub-directory within the project named 'node_modules'. Adding the '--save' flag will save the

dependency name and version of the dependency installed to a file called 'package.json'. This allows using the command 'npm install' which will install all dependencies from the package.json that are missing from 'node_modules' directory. The '--save-dev' flag adds the module to 'devDependencies' in the 'package.json'. This distinction shows which modules are required for the project to be run, and which are required to develop for the project.

5.1.2 Bower

Bower is a package manager similar to NPM. Bower's focus is instead on client-side libraries, and modules. It follows all the same conventions, and has similar flags for installation. To install bower modules use the same command in Figure I, but with 'bower' instead of 'npm'. Bower modules are installed in a sub-directory 'bower_components'.

5.1.3 Gulp

To enable quick, easy and repeated deployment and testing of the code-base the students used GulpJS, and plugins designed for gulp to automate tasks.

```
1.  gulp.task('scripts', ['js-lint'], function() {  
2.    return gulp.src(paths.scripts)  
3.      .pipe(ngAnnotate())  
4.      .pipe(sourcemaps.init())  
5.      .pipe(concat('echo.js'))  
6.      .pipe(wrap('(function (){\n"use strict";\n <%= contents %>\n})();'))  
7.      .pipe(uglify())  
8.      .pipe(sourcemaps.write())  
9.      .pipe(gulp.dest('public/javascripts'))  
10. })
```

Annotation: Figure III: A Gulp task.

```
1.  gulp.task('clean', function (cb) {  
2.    del(paths.clean, cb)  
3.  })
```

Annotation: Figure IV: A Gulp task with a callback.

Gulp's code is divided into tasks as shown in Figure I. These tasks are stored in a gulpfile.js file stored in the root of the project directory. The task must either have a return statement or provide a callback as shown in Figure II. This is required as gulp (and

to a greater extent Node.JS) is designed to be asynchronous. Being asynchronous allows multiple tasks start simultaneously allowing for quicker build times. However tasks can have dependencies (or requirements) before their task is run. As seen in Figure 1 on the first line an Array is added into the task with a string of a name of another task. This indicates to gulp to run and complete that task before running this one. In this case the 'scripts' tasks needs the 'js-lint' (which is a task that lint's the JavaScript ensuring it's valid) before it runs its task. A Gulp task runs in the following manner.

Methods are specified as such:

```
1.      object.methodName(ParameterName{type}, [OptionalParameter])
```

```
1.  gulp.task(Name{String}, [Dependencies{[String]}], [Task{Function}])
```

Specifies the task name, dependencies, and function to execute after the dependencies have completed. A gulp task can simply contain a list of dependencies needed to run, and not have a function to run.

```
1.  gulp.src(Path{String})
```

Specifies the path of the files. This can include Wildcard [26] syntax. The file is then copied, turned into a Stream [27] object, and kept in memory.

–All methods following are chained when used. E.g. `gulp.src().pipe()`

```
1.  gulp.pipe(Plugin{Function})
```

The Stream object is then passed into the function to be changed. How the file's content is changed is dependent on the function.

```
1. gulp.dest(Path{String})
```

Placed inside `gulp.pipe()` so it can access the Stream object containing the modified contents of the file. The Stream is then written to a file at the path specified, and cleared from memory.

```
C:/User/../../Echo> gulp taskName
```

Annotation: Running a gulp task.

To run a Gulp task, gulp must be installed globally. Once gulp is installed globally use the command shown in figure V. having a task called 'default' will allow calling just 'gulp' in your project directory, this will call the 'default' task not every task in the `gulpfile.js`.

5.1.4 Gulp Plugins

gulp-uglify: Takes the JavaScript file and compresses it to the smallest size possible. This is done by removing all whitespace, converting variable names to single letter characters E.g. 'var variable' turns into 'var a'. Most if statements are converted into the ternary operator [28]. This keeps the file size very small, allow for quick load times.

gulp-concat: takes all files from the given path and compresses them into a single file with a name specified. This reduces the amount HTTP requests to one file per file type. E.g. one for all JavaScript files, one for all CSS files.

gulp-sourcemaps: Source maps, map minified code from gulp-uglify to the original source code. This is purely for development to allow for easy debugging, while using the small amount, and small size HTTP Responses.

gulp-ng-annotate: AngularJS has a problem with minified code. Angular normally parses through the parameters of the function in the Controller/Service (See AngularJS Section). This module injects the parameter names as strings to the Controller/Service so it can be preserved after minification.

gulp-jshint: JSHint is a JavaScript Linter it parses through the code and detects errors, common mistakes, or just style choices and outputs any found to the console. The project also uses 'jshint-Stylish' which formats the console output to make it clearer which lines and pages the errors are located.

gulp-nodemon: nodemon is a utility that monitors files for changes and restarts the node server on those changes. gulp-nodemon can also run gulp tasks on restarting the project uses nodemon to 'recompile' everything needed for development, and restart the node server.

gulp-util: gulp-util allows the students to pass in command line arguments to the gulpfile.js. This is used to define the trace level of the server, define whether to lint the server code or the client side code, and to define where the MongoDB executable is located should it not be placed in the default.

gulp-sass: gulp-sass uses the node-sass compiler to compile the Sass to CSS.

gulp-minify-css: minify-css uses the same process as uglify, but is instead geared towards CSS.

gulp-imagemin: imagemin compresses images. The level of compression can be changed and is also dependent on the running computer's processor speed.

```
.pipe(wrap('(function (){\\n "use strict";\\n <%= contents %>\\n})();'))
```

Annotation: gulp-wrap

gulp-wrap: gulp-wrap takes the file(s) given and places them within content specified as shown in figure VII. This is used in the project to make wrap the code in an IIFE (Immediately Invoked Function Expression), and a "use strict"; this scopes our code inside the function hiding the variables from the browsers console, and prevents the variables from polluting the global namespace. "use strict"; is used to tell the browser to place harsher limits on the code, like making silent errors print out an error.

del: del is a core node module. It deletes files, and directories in the paths specified. This is used to clean the build folders to the folders aren't polluted with old files.

child_process: child_process is a core node module. This project uses the exec function of child_process. exec executes a command specified as a string in a new command window.

Gulp tasks:

clean: deletes all files, and folders specified.

Js-lint: passes the files to the JSHint plugin.

Scss: gets a sourcemap on the sass files, concatenates them into a single 'style.css', this file is then minified using minifyCSS. The Sourcemap is placed at the bottom of the file, and the file placed into the build folder.

Scripts: depends on js-lint. The angular dependencies in the files are injected using ngAnnotate. The sourcemaps are then retrieved on the files. The files are then concatenated into 'echo.js' and wrapped in IIFE. The sourcemaps are then written to file and the file is written to the build directory.

imagemin: gets all images, and compresses them and places the new compressed images in the build folder.

demon: starts the node server, and adds a debug level if provided. On start and restart of the server nodemon calls the 'recompile' task.

mongo: starts another process to start the MongoDB Server. This current command will only work for windows, but neither of the students used another OS during the course of the project.

Recompile: a wrapper task calling clean, scss, scripts, imagemin

default: Wrapper task calling demon task, and mongo.

5.2 SERVER INFRASTRUCTURE

Node.JS is used build on the User structure, interact with the database through a REST API, and handle peer to peer communication.

5.2.1 Technology/Library Summary

Express [29]: A web framework for Node.JS.

MongoDB [30]: A NoSQL /document based database.

Body-parser [31]: Middleware from Express to parse HTTP Requests into JSON.

Compression [32]: Middleware from Express to compress HTTP Responses.

debug [33]: Debugging utility for Node.JS.

jsonwebtoken [34]: module that implements JSON Web Tokens[s] for Node.

Mongoose [35]: A MongoDB object modeling module.

Morgan [36]: Middleware from express to log information about HTTP Requests.

serve-favicon [37]: Middleware from Express to serve a favicon.ico

5.2.2 NodeJS

```
1. var element = document.getElementById('id')
2.   element.colour = 'black'
3.   element.innerHTML = 'Hello World'
```

Annotation: Figure IX: Linear coding. (Pseudo Code)

```
1. document.getElementById('id', function (element) {
2.   element.colour = 'black'
3.   element.innerHTML = 'Hello World'
4. })
```

Annotation: callback coding. (Pseudo Code)

All node code is written using “callbacks” this means having a function that is called at the end of another function rather than having linear code style as shown in figure IX and X. The function placed within the parameters is executed at the end of the function. This means code written after the call can't rely on the data desired to be there. All code relating to the data must be contained within the callback. While at first this seems very constricting in practice it allows the code to be very high performance, and keep all connections on a single thread.

5.2.3 Express

Express is used to abstracting configuring and designing a web server. We set our views to the 'jade' folder, and set the view engine to jade. Jade is a rendering template designed for express a sample comparison is shown in figure XI and XII. The view engine setting tells express we are using a rendering template like jade.

```
1.  html
2.    head
3.      title Example Jade
4.    body
5.      h1 Hello World
6.      p.className This is a p tag
```

Annotation: Jade sample

```
1.  <html>
2.    <head>
3.      <title>Example HTML</title>
4.    </head>
5.    <body>
6.      <h1>Hello World</h1>
7.      <p class="className">This is a p tag</p>
8.    </body>
9.  </html>
```

Annotation: Jade sample in HTML

```
1.  app.use(function (req, res, next) {
2.    res.setHeader('Access-Control-Allow-Origin', '*')
3.    res.setHeader('Access-Control-Allow-Methods', 'GET, POST', 'PUT', 'DELETE')
4.    res.setHeader('Access-Control-Allow-Headers'
5.      , 'X-Requested-With, content-type, Authorization'
```

```
6.      )  
7.    next()  
8.  })
```

Annotation: An Express middleware function.

5.2.4 Server Configuration

`app.use()` gives express middleware to apply to incoming requests and outgoing responses. All middleware functions have three parameters. Request, Response, and Next. Request is an object representing the information about the request sent. Response is an object representing the response to the request. Next is called at the end of the middleware to allow the next middleware/function to access the request. All middleware manipulates the request, and response objects.

Using the compression library all responses are compressed using zlib (zlib is a compression algorithm [38]). Using the serve-favicon library we provide a path to the projects favicon.ico. Favicon (short for favorite icon) is a small icon used by web browsers and mobile devices when bookmarking, or saving the web page. Using the body-parser library we allow data sent with the request be sent through JSON, or through an encoded URL. The data is then attached to a 'body' property which is then attached to the request object allowing all following middleware to access the data. The project has 3 static routes. Everything generated in the public directory is given its own route E.g. `/javascripts/echo.js`. A route dedicated to images E.g. `/img/favicon.ico`. The final route is for libraries obtained through bower e.g. `/bower_components/angular/dist/angular.min.js`.

The final main middleware we use attaches three headers to all responses. 'Access-Control-Allow-Origin: *' this allows any device to make calls to our API. 'Access-Control-Allow-Methods: GET, POST, PUT, DELETE' this allows only get, post, put, and delete methods to be accepted. 'Access-Control-Headers: X-Requested-With, content-type, Authorization' this allows the project to have authorized routes without using cookies.

5.2.5 REST API

For detailed information on individual API routes open the `'docs/index.html'` file in the project directory. The API is centered on getting users (And messages before it was cut). Using the API the project can sign in a user to the service, get their friends, update their details, delete them, add, remove, and accept their friends, and validate the users' token. When a user gets other user's all non-essential details about the user are copied into a new object and then removed and sent to the user, this prevents people getting vital information about user like their password through the console. Some routes require authorization in order to be used. For authorization the User model contains a

property called token which contains the last token generated. These tokens change every time the user data has changed. This means that if a user's computer has a token that was before the user updated their details they will be denied access and have to login again. All responses send back a JSON object with a token property containing the updated token.

```
1. module.exports = function (req, res, next) {  
2.   var authHeader = req.get('authorization')  
3.   if (typeof authHeader === 'undefined'){  
4.     res.status(403).end()  
5.   }  
6.   else {  
7.     var authValue = authHeader.split(' ')  
8.     , token = authValue[1]  
9.  
10.    debug('Token: ', token)  
11.    User.findOne({token : token}, function (err, user) {  
12.      if (err)  
13.        res.send(err)  
14.  
15.      if (!user) {  
16.        res.status(404).end()  
17.      }  
18.      else {  
19.        req.user = user  
20.        debug('User added to request:\n', req.user)  
21.        next()  
22.      }  
23.    })  
24.  }  
25. }
```

Annotation: Authorization middleware.

How the authorization works is by getting the authorization header in the request. If the header is not there a 403: *Forbidden* error is sent. The token value is then extracted from the header and then is queried against the database. If the user is not found the middleware sends back a 404: *Not Found* error. If the token is valid the user attached to that token is then attached to the request object and pass along the route that required the authorization.

5.2.6 MongoDB

```
1. var mongoose = require('mongoose')  
2. , Schema = mongoose.Schema  
3.  
4. module.exports = mongoose.model('User'
```

```
5.         , new Schema({ email : String
6.           , username : String
7.           , password : String
8.           , friendsList : [String]
9.           , pendingList : [String]
10.          , avatar : String
11.          , token : String
12.          })
13.      )
14.  )
```

Annotation: User Schema.

For connecting to MongoDB, this project uses mongoose a library designed to model MongoDB's Documents to JSON objects. For the user data a model, is needed to be defined. A model contains the name to describe the data E.g. User, and a Schema which defines the properties, and their type. One of the main advantages of MongoDB is the Array{Type} type. Where relational databases like SQL the project's user model is would need three tables to handle its two many to many recursive relationship. In MongoDB a document can contain an array which can store references to other documents. This is used for the user's pending list (For when a user has added a friend, but the friend hasn't accepted their request), and the user's friends list (When both a user and the friend have accepted each other).

5.2.7 Socket.IO

For handling user's calling each other (and for notifications, and messages before they were cut). The project uses Socket.IO which is a library that uses WebSockets (and AJAX requests if WebSockets isn't supported) to communicate peer to peer. When a user successfully logs in an event is emitted placing them inside a room with their own MongoDB ID (On creation of a document/user MongoDB creates a BSON property based on different factors to ensure to uniqueness). Having a user in a room allows us to discriminate messages being sent between users. Normally when an event is emitted it is emitted to all sockets. If a user is in a room, however the server can send an event just to them. This allows users to call each other without worrying about the wrong user getting called.

5.2.8 Web RTC

For the development of the project a web RTC library was used. However the following section will explain how the basic web RTC concepts work.

For getting the user's microphone, the getUserMedia API is used which provides a MediaStreamTrack [39] object which can be transferred across connections or passed in an HTMLVideoElement to be displayed on the screen. In order to use this API. When

the `getUserMedia` is first called, the scripted is blocked, and the user is prompted by the browser whether they wish to provide permission to the web application. This permission is typically for one use, and permission will be required again once the page has been refreshed.

For transferring the users `MediaStream` to the other user, the `PeerConnection` API is used. The `PeerConnections` offsets the heavy traffic of transferring audio, and/or video streams, across the internet to the clients who are requesting and sending the data. This allows for the servers for the application to focus on the relatively lightweight requests of getting web resources and communication with the database.

Once a request for a call has been sent a user is then alerted to the request, based on the user's response a 'declined' event, an 'accepted' event is emitted. If a call is accepted the `SimpleWebRTC` [40] is started. For working with the framework we provide the elements to attach the user's local stream. And place them into a room for STUN (Session Traversal Utilities for NAT [Network Address Translation]) server. A STUN server finds the user's real IP address endpoint when they are behind NAT.

However STUN does not cover "Symmetric NAT" which is when a unique port is placed on each connection to the server from the same device [41]. The STUN servers used for this project are provided by the creators of `SimpleWebRTC`. To prevent the "Symmetric NAT" problem TURN (Traversal Using Relays around NAT) are used. If the STUN server has failed to connect the client. The request is then sent to the TURN server which then provides a connection between just the client and TURN server. The TURN server, on processing the information sent by the client then relays that information to the peer, the client was attempting to communicate to [42].

5.3 USER INTERFACE

Echo is a full screen web page application designed for desktop and laptop use primarily. A responsive mobile first layout is not being used as the technology behind the app is currently not supported on a number of key mobile operating systems [43].

The structure of the website is based around the idea of a single page application (SPA). From when the user initially loads the page to when they close it, the page never reloads once. All this is done using angularJS.

To speed up the workflow of the project we used jade (a templating engine for html) and SASS (an upgraded CSS with variables etc.).

Skeleton [44], a CSS boilerplate, was used as the structure for the page. Skeleton is a lightweight boilerplate, containing approximately 200 lines of code that uses the twelve column grid system for laying out a page. This allowed the content of the site to be easily organized into easy to manage columns. Skeleton does have a responsive layout so some of the web page does change with the size of the screen, however many of the elements are not designed for this layout. Skeleton uses two files, skeleton.css, which gives the styling to certain elements and creates the grid. The second file is normalize.css, which removes any of the styling applied by the browser.

5.3.1 Screens

5.3.1.1 Login

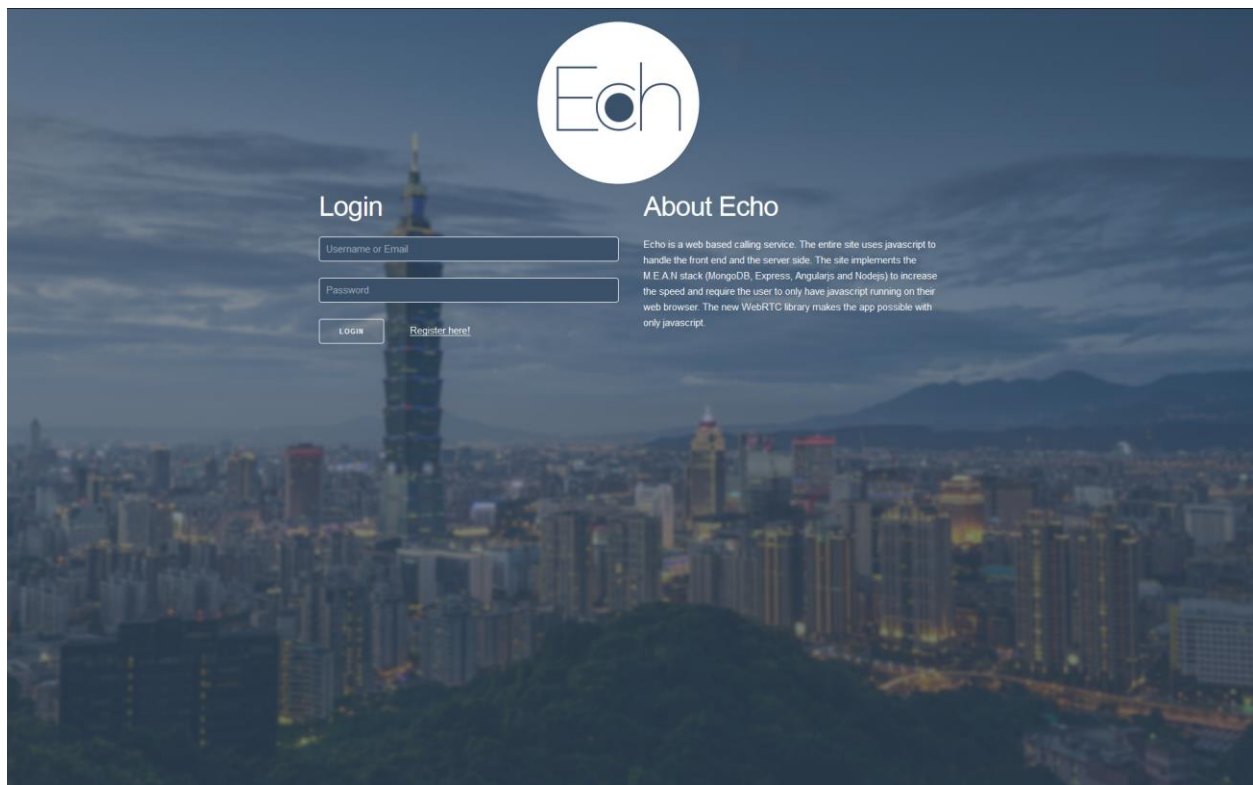


Figure 6.1 - Login Screen Final Version

The login screen is the first screen the user is brought to. The login form requires the users email and password, beside the form is a small paragraph about the app. These contents are in a 960px wide container (skeleton default size) and both the form and the paragraph are set to six columns each or 50% of the container. The logo of the app is placed above, center aligned. A full vertical height image is the background.

The form uses angular modules for validation. The email form check if the user has written any text to the input box and if so then check if the text entered is a valid email address. The input also checks if the box is blank. If any of these errors are met then an

error message appears in a span tag below the input box. The password box uses a similar validation method except instead of checking for an email, it checks what the length of the password is. If the password is shorter than 5 characters or longer than 30 characters an error message appears. If any of the above errors are met then the submit button at the end of the form is disabled and the user cannot submit the form. Once the errors are fixed the button is enabled for submission.

```
1. form(name="loginForm" ng-controller="FormController as form")
2.   ul
3.     li
4.       input.u-full-width(name="loginEmail" type="email" placeholder="Username or Email" ng-
5.         model="user.email" required novalidate autofocus)
6.       span(ng-show="form.errorStatus === 404") User not found!
7.       span(ng-
8.         show="loginForm.loginEmail.$dirty && loginForm.loginEmail.$error.required") Required!
9.       span(ng-
10.        show="loginForm.loginEmail.$dirty && loginForm.loginEmail.$error.email") Not a valid email!
11.     li
12.       input.u-full-width(name="loginPassword" type="password" placeholder="Password" ng-
13.         model="user.password" ng-minlength="5" ng-maxlength="30" novalidate required)
14.       span(ng-
15.         show="loginForm.loginPassword.$dirty && loginForm.loginPassword.$error.required") Required!
16.       span(ng-
17.         show="loginForm.loginPassword.$dirty && loginForm.loginPassword.$error.minlength") Password too
18.         short!
19.       span(ng-
20.         show="loginForm.loginPassword.$dirty && loginForm.loginPassword.$error.maxlength") Password too
21.         long!
22.     li
23.       input(type="submit" class="button" ng-disabled="loginForm.$invalid" ng-
24.         click="form.submit(user)" value="Login")
25.       a(href="#/register") Register here!
```

5.3.1.2 Register

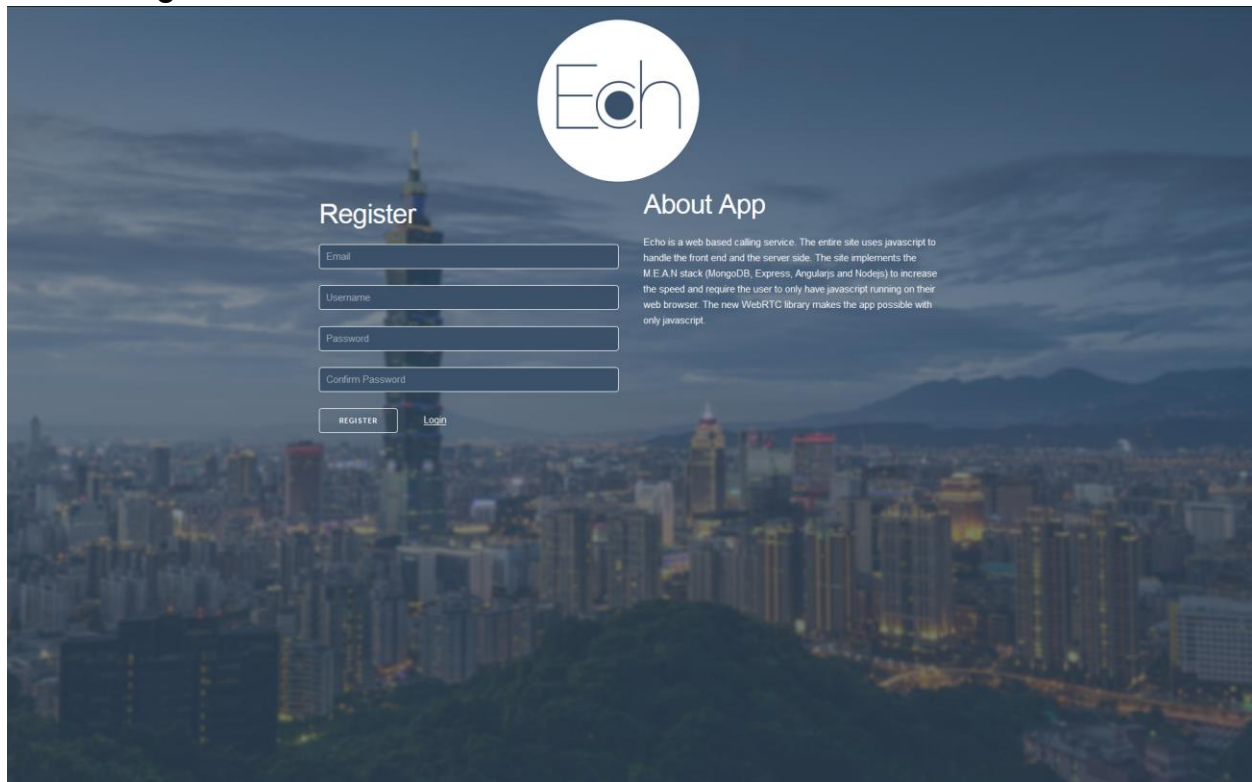


Figure 6.2 - Register Screen Final Version

The registration screen is almost identical to the login screen except for the form having two more inputs. Requiring a username and a password confirm. The username input uses the same validation as the password fields, that the username must be longer than 5 characters. The password confirmation uses a custom made angular module[45] which checks that the contents of the password field matches the contents of the confirm password field. Just like the login form if any of the errors are met then the submit button is disabled until the errors are fixed.

5.3.1.3 Dashboard

Once signed in the user is brought to the dashboard. This is the main screen for the majority of the time spent in the app. The user's friends are displayed on the left side of the screen in two columns. The rest of the content of the page is in the remaining ten columns. In the ten column section there are three rows: the navigation bar at the top of the screen, the first row which contains user info and the second row which displays the call when the call is running.

The design for the dashboard was the result of research of the different social networking sites and other calling services such as skype and Facebook.

5.3.1.3.1 Initial Login

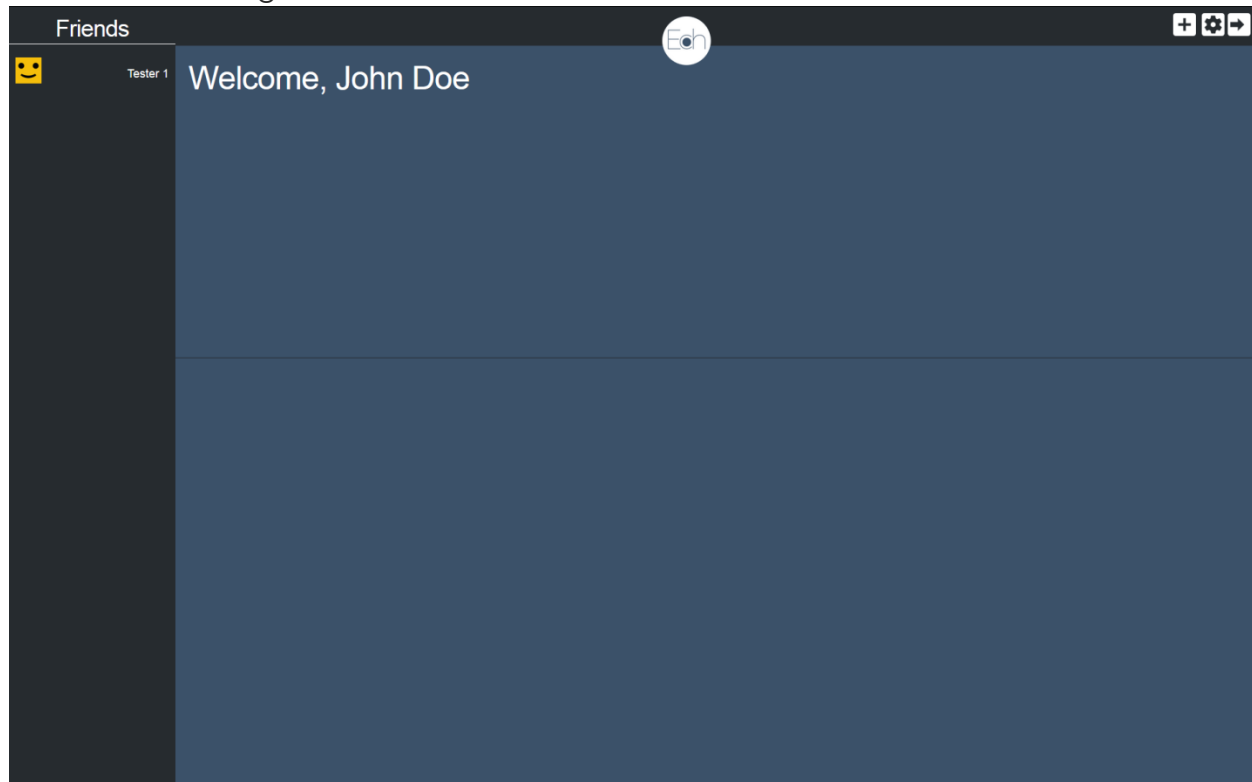


Figure 6.3 - First Dashboard Screen

The dashboard screen when the user first logs in shows only their friends in the friends list and in the first row a small welcome message. The second row remains blank. In the navigation bar there are three buttons, all are SVG's (Scalable Vector Graphics). The first is the button to add a user. Once clicked an input box appears with a confirm button, and cancel button. An error has occurred with the input field as some styling has pushed it out of position due to styling taken from skeleton. The input field and confirm/cancel buttons are hidden using the angular module ng-show, once the add button is clicked they are displayed using ng-show.

```
1.      li
2.      input.addUser(type="text" placeholder="friends email" ng-show="dash.showAddbar" ng-
      model="queries.addQuery")
3.      img.addUser(ng-src="/img/correct.svg" ng-show="dash.showAddbar" ng-
      click="dash.addFriend(queries.addQuery); dash.showAddbar=false" alt="Add")
4.      img.addUser(ng-src="/img/delete.svg" ng-show="dash.showAddbar" ng-
      click="dash.showAddbar=false" alt="Cancel")
```

Beside the add button is the button to change the users settings, this is a link that brings a user to a form which allows them to change their account details. Last is the logout button which sends the user to the login screen and clears the users token.

5.3.1.3.2 Friend Clicked

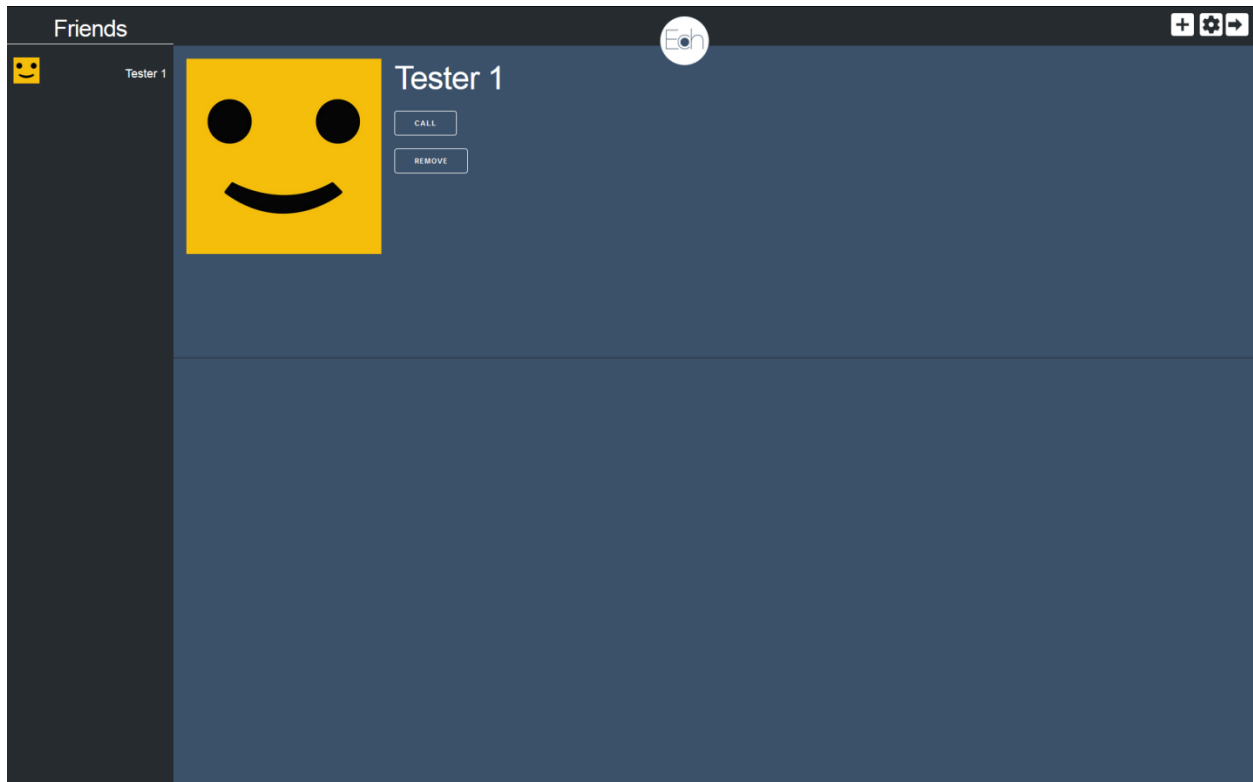


Figure 6.4 - Dashboard View when Friend Clicked

Once a user in the friends list is clicked the contents of the first row is changed. The welcome message is cleared and the friend's details are displayed. From this display the user is now able to call another user or remove them as a friend.

5.3.1.3.3 User Calling

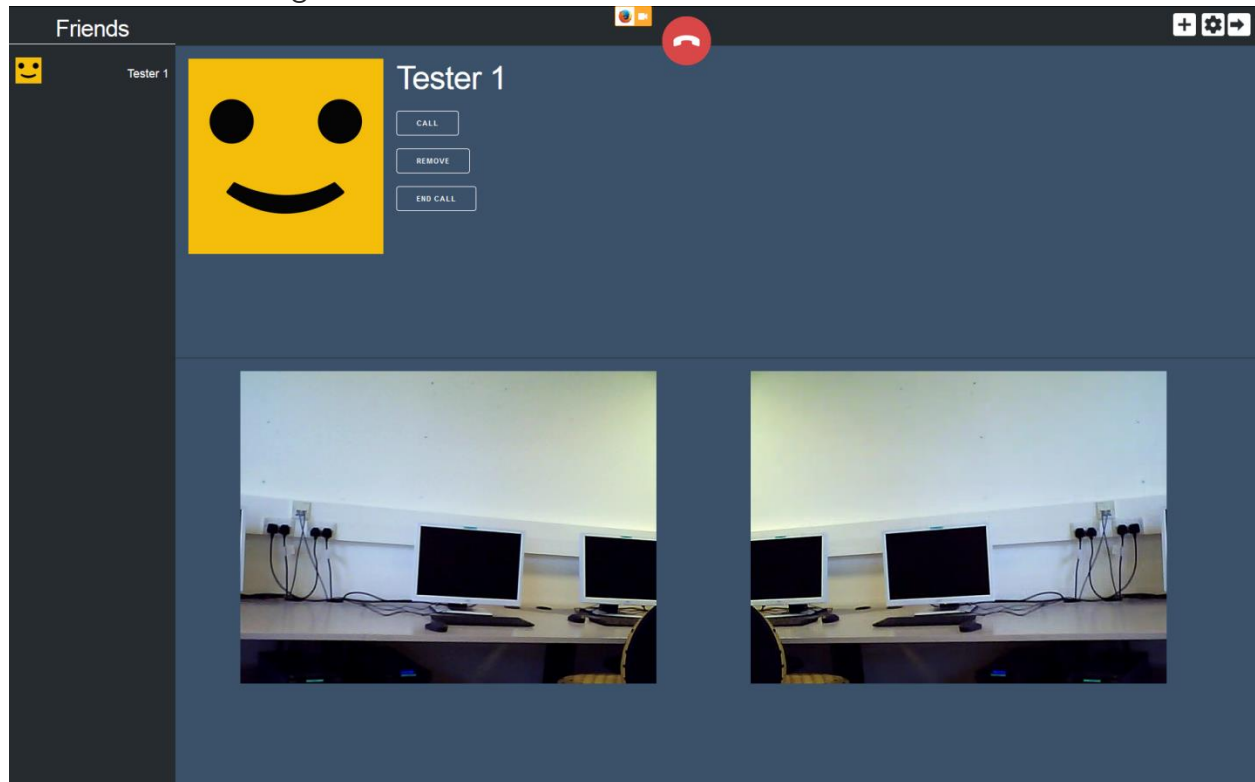
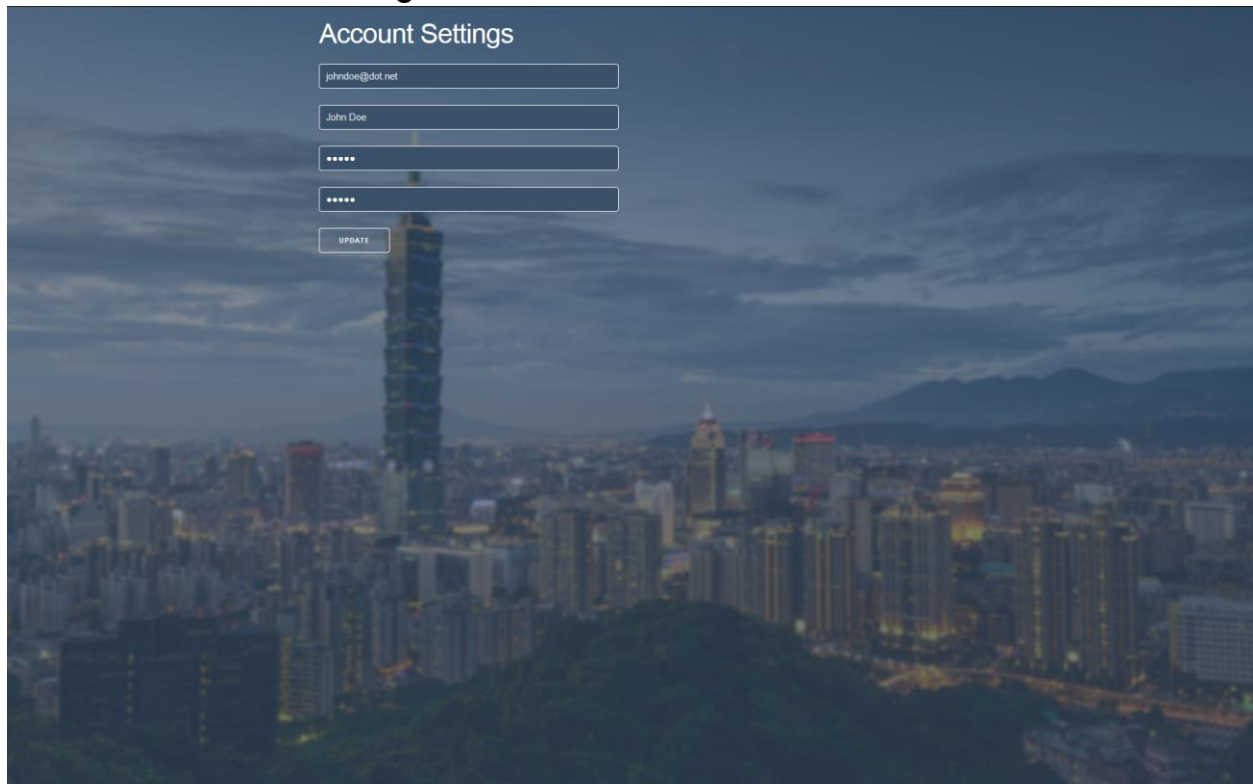


Figure 6.5 - Dashboard Screen during Call

Once the button is pressed to call a user, two video elements display in the second row. Both are set to 640x480 pixels. The left video is the local video (you) and the right video is the remote (friend). An end call button appears in place of the app logo at the top center of the page. This only appears when a call is started and disappears when the call ends.

5.3.1.4 User Account Settings



The image shows a web application interface for 'Account Settings'. The background is a dark, high-angle photograph of a city at night, with the Taipei 101 skyscraper prominently visible. Overlaid on this background is a white form with the title 'Account Settings' at the top. The form contains four input fields: the first is for an email address (pre-filled with 'john.doe@dot.net'), the second is for a name (pre-filled with 'John Doe'), the third and fourth are for passwords (both pre-filled with six dots). Below these fields is a small 'UPDATE' button.

Figure 6.6 - Account Settings Screen

When a user clicks the link to edit their account details they are brought to the account details screen. The screen contains a form, the same as the register screen, which has their details filled in. Here the user can edit the details in the input fields. Once their details are updated they submit the form and are brought back to the dashboard. If the user wants to cancel the update they can head back to the dashboard by pressing their browsers back button.

6 TESTING

While no formal user testing, or unit testing was carried out. The students would create test user accounts, and test the system against those accounts. The API was also tested using a Google Chrome application called Postman. This allowed the students to repeatedly test different requests, and to stress test the RESTful API. This allowed the API to be refined, and have the number of routes used reduced.

7 CONCLUSION

7.1 EVALUATION

Over the course of the project we learned how to build data driven applications using AngularJS. With two way data binding, having the data that was received from the back-end reflected in the user interface allowed us to build a high performant application. Using Node.JS, and socket.IO, we learned how to deliver server-side information in real time. MongoDB taught us different types of Database designed, and allowed us to design the database, in entirely different way than how we were taught in college.

The Goal of this project was to create an application that allows users to call one another in a service based in the web browser. This goal was accomplished but a number of different features were not completed.

To accomplish this goal a number of techniques were used:

- Weekly meetings with our supervisor - Receiving feedback on a consistent basis allowed us to remain on track throughout the year.
- Setting deadlines – On a week by week basis we set a number of deadlines for each other to complete throughout the week. This prevented the possibility of getting distracted by another part of the project.
- Splitting up the workload – Like setting deadlines this helps prevent the problem of getting distracted by other parts of the project.

A number of problems did appear during development. The biggest issue was the decision about Web RTC. Initially we created our own Web RTC but quickly realized that a number of libraries existed to help speed up development. Once the decision was made to use a library the next problem was what library to use. A large number exist and the choice of library changed a number of times throughout development. Some libraries were well developed but had not been updated for a considerable time. Others were maintained frequently but did not fit the role needed.

The biggest omission in the final version of the app was the messaging system. Time was a significant factor in making the application, of all the key elements of the app, messaging took a back seat to other more important features, such as calling and the friends list. As a result of this, the messaging was left until last and was not implemented.

7.2 FUTURE WORK

A number of elements of the app could be improved upon if more time was applied to it in the future.

First and most importantly is the messaging system. The messaging system simply needed more time to implement. It is a key feature in most web based communication and it only makes sense for it to be in place here.

Secondly a mobile version of the app would be a mandatory inclusion in the future. At the moment Web RTC is still not supported by all browsers, most notably safari for mac and iOS. With the inevitable support for Web RTC in the future a mobile view for the app would greatly improve usage on mobile devices, thus vastly increasing the number of users.

Finally a bigger focus would be spent on the design of the apps visuals. The timeframe for the project was of a decent length but quite a bit of that time was focused on the core functions of the project working. UI/UX is a very important part of any web app/site and needed more time to be fully fleshed out. Lastly, more time spent on the design gives a more polished feel to any project, small additions such as animations can really finish off the look of an app.

8 REFERENCES

[1]: ECMAScript (N.D) Retrieved November 30th 2014, from ECMA International

<http://www.ecmascript.org/>

[2]: Kangax, Webbedspace, ECMAScript Compatibility Table, 29th November 2014, Retrieved on November 30th 2014, from GitHub Inc.

<http://kangax.github.io/compat-table/es5/>

[3]: node.js, (N.D), Retrieved November 30th 2014, from Node.js project, Joyent, Inc.

<http://nodejs.org/>

[4]: List of contributors by first contribution. Published on September 17th 2014, Retrieved November 30th 2014, from Github, Inc.

<https://github.com/joyent/node/blob/master/AUTHORS/>

[5]: Projects, Applications, and Companies Using Node, Published on November 18th 2014, Retrieved November 30th 2014, from Github Inc.

<https://github.com/joyent/node/wiki/Projects,-Applications,-and-Companies-Using-Node>

[6]: Express - Node.js web application framework, (N.D), retrieved on November 30th 2014, from StrongLoop.

<http://expressjs.com/>

[7]: MongoDB – The Leading NoSQL Database | MongoDB, (N.D), Retrieved November 30th 2014, from MongoDB Inc.

<http://www.mongodb.com/leading-nosql-database/>

[8]: Node Streams: How do they work?, Max Ogden, April 2012, Retrieved November 30th 2014, from Max Ogden's Blog, Freelance.

<http://maxogden.com/node-streams.html>

[9]: Node.js v0.10.33 Manual & Documentation, (N.D), Retrieved November 30th 2014, from Node.js project, Joyent Inc.

<http://nodejs.org/api/stream.html>

[10]: WebRTC FAQ, (N.D), Retrieved November 30th 2014, from Google Sites, Google, Inc.

<http://www.webrtc.org/faq/>

[11]: WebRTC, ryunosuke_OS, rvighne, chrisdavidmills, teoli, AlisaRivera, Ayana_Taco, November 19th 2014, retrieved November 30th 2014, from Mozilla Developer Networks, Mozilla.

<https://developer.mozilla.org/en-US/docs/Web/Guide/API/WebRTC/>

[12]: RTCPeerConnection, Delapouite, Developedbyme, Pearlmutter, kscarfone, teoli, Yunzhou, wesj, November 27th 2014, retrieved November 30th 2014, from Mozilla Developer Networks, Mozilla.

<https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection/>

[13]: Navigator.getUserMedia, Sheppy, scottrowe, juhsu, openjck, pdm, MattBrubeck, ethertank, merih, kscarfone, adambom, teoli, tagawa, chrisdavidmills, August 18th 2014, Retrieved November 30th 2014, from Mozilla Developer Networks, Mozilla.

<https://developer.mozilla.org/en-US/docs/NavigatorUserMedia.getUserMedia/>

[14]: RTCDataChannel, tOkeshu, teoli, August 4th 2014, retrieved November 30th 2014, from Mozilla Developer Networks, Mozilla.

<https://developer.mozilla.org/en-US/docs/Web/API/RTCDataChannel/>

[15]: Test the new Firefox hello webRTC feature in Firefox, Chad Weiner, October 16th 2014, retrieved December 4th 2014, from Mozilla's Blog.

<http://blog.mozilla.org/futurereleases/2014/10/16/test-the-new-firefox-hello-webrtc-feature-in-firefox-beta/>

[16]: Tokbox pricing, (N.D), Retrieved December 4th 2014, from Tokbox.

<https://tokbox.com/pricing/>

[17]: Merged pull request, March 6th 2014, retrieved November 30th 2014, from Github Inc.

<https://github.com/webRTC/webRTC.io/commit/e0d11fe43ae0a4fa1d7c4f853d487427b318acf3/>

[18]: GitHub, (N.D), retrieved March 25th 2015, from GitHub

<https://github.com/>

[19]: Git, (N.D), retrieved March 25th 2015, from Git

<http://www.git-scm.com/>

[20]: Github Commits, (N.D), retrieved March 26, 2015, from Github

<https://github.com/Aaronepower/Echo/commits/master>

- [21]: Google V8, (N.D), retrieved March 26, 2015, from Google Code
<https://code.google.com/p/v8/>
- [22]: NPM, (N.D), retrieved March 26, 2015, from NPM
<https://www.npmjs.com/>
- [23]: Bower, (N.D), retrieved March 26, 2015, from Bower
<http://bower.io/>
- [24]: Gulp, (N.D), retrieved March 26, 2015, from GulpJS
<http://gulpjs.com/>
- [25]: Node, (N.D), retrieved March 26, 2015, from Node
<https://nodejs.org/api/index.html>
- [26]: Wildcard syntax, (N.D), retrieved March 26, 2015, from MSDN
https://msdn.microsoft.com/en-us/library/ms171453.aspx#BKMK_Wildcards
- [27]: Node Stream API, (N.D), retrieved March 26, 2015, from Node
<https://nodejs.org/api/stream.html>
- [28]: JavaScript, (N.D), retrieved March 26, 2015, from Wikipedia
<http://en.wikipedia.org/wiki/%3F:#JavaScript>
- [29]: Express, (N.D), retrieved March 26, 2015, from Express
<http://expressjs.com/>
- [30]: MongoDB, (N.D), retrieved March 26, 2015, from MongoDB
<http://www.mongodb.org/>
- [31]: Body Parser, (N.D), retrieved March 26, 2015, from Github
<https://github.com/expressjs/body-parser>
- [32]: Compression, Commits, (N.D), retrieved March 26, 2015, from Github
<https://github.com/expressjs/compression>
- [33]: debug, (N.D), retrieved March 26, 2015, from Github
<https://github.com/visionmedia/debug>
- [34]: Json Web Tokens for Node, (N.D), retrieved March 26, 2015, from Github
<https://github.com/auth0/node-jsonwebtoken>
- [35]: Json Web Token Specification, (N.D), retrieved March 26, 2015, from Github
<http://self-issued.info/docs/draft-ietf-oauth-json-web-token.html>
- [36]: Mongoose, (N.D), retrieved March 26, 2015, from Mongoose
<http://mongoosejs.com/>

- [37]: Morgan, (N.D), retrieved March 26, 2015, from Github
<https://github.com/expressjs/morgan>
- [38]: Serve Favicon, (N.D), retrieved March 26, 2015, from Github
<https://github.com/expressjs/serve-favicon>
- [39]: zlib, (N.D), retrieved March 26, 2015, from zlib <http://www.zlib.net/>
- [40]: SimpleWebRTC, (N.D), retrieved March 26, 2015, from SimpleWebRTC
<https://simplewebrtc.com/>
- [41]: STUN servers, (N.D), retrieved March 26, 2015, from voipInfo <http://www.voip-info.org/wiki/view/STUN>
- [42]: TURN Server Specification, (N.D), retrieved March 26, 2015, from IETF
<http://www.ietf.org/rfc/rfc5766.txt>
- [43]: MediaStreamTrack, (N.D), retrieved March 26, 2015, from MDN
<https://developer.mozilla.org/en-US/docs/Web/API/MediaStreamTrack>
- [44]: getUserMedia support, (N.D), retrieved March 26, 2015, from Caniuse
<http://caniuse.com/#search=getusermedia>
- [45]: Skeleton Framework, (N.D), retrieved March 26, 2015, from skeleton
<http://getskeleton.com/>
- [46]: Angular input match module, (N.D), retrieved March 26, 2015, from ngmodules
<http://ngmodules.org/modules/angular-input-match>