**Creating a Social Media App Focused on Privacy**

By Elliot Holbrook, Queen Elizabeth’s Grammar School [017080]

**Contents**

Analysis

* Introduction
* Computational Amenability
* Stakeholders
  + Gamers
  + Students
  + Individuals That Are Privacy Conscious
* Research
  + Existing Implementations
    - Masterdon
    - MeWe
  + Conclusions
    - Positives
    - Negatives
* Requirements For the Proposed Solution
  + Development Requirements
  + End User Requirements
* Limitations Of The Proposed Solution
  + Security
  + Servers
  + Features
  + Bugs
* Success Criteria
  + Simplicity
  + Modern User Interface
  + Messaging
  + Privacy
  + Hubs and Interests

Design

* Methods, Variables, Classes and Parameters
* Algorithms
  + Flowcharts
    - Displaying History of a Chat to a User
    - Allowing a User to Send a Chat
    - Allowing a User to Log In
    - Blocking a User
  + Validation Methods
    - General Inputs
    - Emails
    - Usernames
    - Profile Pictures
    - Passwords
    - Pins
  + Pseudocode
    - Friend Requests
    - Deleting a Chat
* Usability
  + Learnability
  + Efficiency
  + Errors and Validation
  + Satisfaction
* Structure of the Application
* Testing
  + General Testing
  + Specific Testing
  + Large Data Set Testing
* Databases
  + Account Data
  + Messages
  + Hubs
* Interface Designs
  + First Ideas

Development

* Accounts and Friends Management System
  + Prototype 1
    - Base Ideas for Accounts and Database Interactions
    - Code Simplification
  + Prototype 2
    - Simplifying and Modularising Database Interactions
      * Create
      * Read
      * Update
      * Delete
    - Log In System
      * Back End
      * Front End
    - Registration System
      * Front End
      * Back End
    - Conclusion of Prototype
  + Prototype 3
    - Addressing Account Object Data Input Validation
    - Ensuring Email Addresses are Unique
    - Using Account Object For Session Identification
  + Prototype 4
    - Storing Friend Requests and Friends in the Database
    - Making and Displaying Friends
      * Displaying Friend Requests
      * Sending and Accepting Friend Requests
* Real Time Chat Service
  + Learning Websockets
    - WebSocket API
    - Socket.io
  + Prototype 1

**Analysis**

**Introduction**

Slowly but surely, data privacy is more and more becoming a luxury rather than a given. The quantity of user data that is secretly collected and sold by large companies, such as Facebook or Google, is astounding and the problem is only getting worse. One of the main sectors that is the culprit of this is the social media industry, who can spy on private conversations and use artificial intelligence to gather more and more data on users. 62% of American adults say that they don’t believe that they can go through daily life without companies collecting data about them (source: Survey June 3rd – 17th 2019 by PEW Research Centre).

This is why I want to make a social media app that focuses on privacy so that people can feel happy about the way their data is handled. I believe that privacy is something that can be restored to the digital landscape, but for this, real working solutions need to be designed and built.

There are already some solutions that mimic the style of existing large social media companies but that focus on data privacy and user safety. There are still some gaps though, where some larger apps do not have privacy centred counterparts. I aim to fill one of these gaps: Discord.

**Computational Amenability**

Social media has already changed the world due to the speed of communication with the technology now being able to send messages and information across the world in seconds.

Messages can only be sent across the world in this time using the existing infrastructure and so it is essential that they are sent digitally through the internet to achieve the same ease of use. It would be completely impractical to communicate through any other method.

My app focuses on privacy, which I believe can be best achieved digitally. The use of modern encryption makes it near impossible to read people’s messages in transit. Any other methods of message transfer, such as post would be much less secure.

It’s much easier to reach a much larger number of people digitally, as 59% of the world’s population now has access to the internet. This is ideal because it’s much easier and more practical to meet people who are interested in the same things as you over the internet. And it also makes it a lot easier to meet friends of your friends who may live geographically quite far away from you.

An open voice chat system is also suits digital communication as well. Unlike a phone call, this would allow any number of users to join a call at any time of the day. Users can leave and join calls when they want with the click of a button.

**Stakeholders**

**Gamers**

This app would be ideal for gamers who want to chat to their friends and organise gaming sessions using large multi-user text chats. It would allow people to set up gaming communities which will allow them to communicate with their teammates and meet new friends that are interested in the same genres as them.

The app is suited to multiplayer gamers because it will be able to run on a browser in the background while they play, allowing them to use voice chats (with very limited delay) to help them win competitions by collaborating closely.

They can then break off into different channels if they decide to split off into subgroups that are no longer playing together.

**Students**

Students, who are interested in growing and excelling in their subjects, would be drawn to the app so that they would not always have to physically meet up with others to discuss topics that interest them.

The existence of chat hubs would allow students create online communities around their disciplines. Hub owners can name their community hubs after these disciplines. This would help them discuss their areas of interest with peers which would aid them in their education but also help them have fun after they have finished their work for the day.

The app will also be free and work with the computer or smartphone that they already own, meaning that using the app will be appealing to users that cannot afford to pay to send messages to others.

**Individuals That Are Privacy Conscious**

A very large proportion of websites track users and their data, so it’s no wonder that so many people are worries about what big companies are doing with their personal data. However, end to end encryption and transparency with the user will be at the forefront of my application. Account security is also very important, and so two factor authentication will be required on all accounts as well as a verified email.

Users will also be able to enter data on who should be able to view their account or send them messages or friend requests so that all users can stay safe and strictly control their own level of privacy.

There will also be the option to add a pin to your account so that you must enter a pin code before you are able to see any messaged that you have previously received or sent or any personal information that you have stored on your account, meaning that even with your login details compromised, you don’t have to worry about your data being stolen.

This group of people if becoming larger and larger as more and more is revealed to the public about how their data is handled. This would give my app a larger and growing userbase.

**Research**

**Existing Implementations**

Mastodon

One existing privacy focused social media app is Mastodon, which is quite similar to Twitter, but it focuses more on user safety and privacy. Mastodon is available on all devices that have access to a web browser and is completely open source and transparent.

The focus on privacy is an important benefit because it has features that allow you to block very specifically who can see your account and who cannot. This enables users to keep safe and I would like to copy this feature so that users will be able to block anybody that causes a threat to them. This would prevent any communication between the two users.

A screenshot of a computer

Description automatically generated with medium confidenceHowever, what I don’t like about Mastodon is that the user doesn’t choose what they are immediately exposed to as they first join the app (as seen in the image below). Instead, for my app, the user will have to seek out hubs that they are interested in, or friends that are already on the app.

I also like how simple the user interface is and will try and keep the user interface of my own app similar to existing social media websites so that transitioning away from them is easier.

I think overall it is quite a successful implementation that may become a rival to Twitter in the future.

MeWe

MeWe is another privacy centered app that is based on trust and the user’s interests, describing itself as “the fun social media experience you can trust, control and love”. It also has a mobile app and is quite accessible for a large range of users and devices.

One good point in favour of MeWe is that is has specific groups for people’s different hobbies and interests which allows people to discuss different topics in small and large communities. This is something that I would like to incorporate into my own app as I think it would benefit the users of the app and increase interaction while making new users more likely to stay.

Graphical user interface, text, application, chat or text message, email

Description automatically generatedHowever, the home screen of the app doesn’t look very modern and it is quite complicated (as seen below) for new users. I believe this is a significant drawback as it is difficult to learn and this could dissuade them from taking to MeWe.

**Conclusion**

Positives

The user should have fine control over the users that can communicate with them and see their account.

There should be a clean and modern look to the user interface that allows for new learners to easily understand how it works.

There should be hubs that the user can join that are based on specific topics.

The app should have a high level of account security including two factor authentication and there should be full transparency with how data is handled.

Negatives

The user interface should not be cluttered and difficult to understand.

The user should not immediately be subjected to content before they have decided that they want to see it.

**Essential Features**

**Messaging**

Messaging is the simplest way to communicate effectively and it is a part of many social media sites, meaning that people are accustomed to using it. Unlike the implementations within the previous solutions mentioned before, it’s also very versatile and can include file sharing/viewing and will make it very clear who will receive the message that you are about to send. I don’t believe that Mastodon does this very well and it would be easy to make mistakes when sending direct messages.

**Directed Communication & Privacy**

Users should be able to have fine control over who their messages and posts can be viewed by and who can communicate with them. This is essential for keeping users safe and happy.

**Expression**

There should access to tools that allow users be able to express themselves on their profile in whatever way they like. This includes things like personal descriptions, usernames and profile pictures.

**Requirements For The Proposed Solution**

**Development Requirements**

I require a computer that can run XXAMP, which will run Apache and MySQL, to function as the web server and database handling server. Ideally I could use a Raspberry Pi to host this as they are low power consumption, can be hooked up directly to a router by ethernet, and require little maintenance, so it could be kept online constantly and I could work on it remotely using a FileZilla file server. This would also allow for testers to connect at any time if I set up port forwarding on the router. This also means that I would need an internet connection and another computer to run a FileZilla client and connect to the Apache server and phpMyAdmin through a browser.

Other than this I will need time to research the programming languages that I will be using as well as time to think about the best way of implementing some of the features.

**End User Requirements**

The user would require simply a computer running a browser and a keyboard to use the app. JavaScript must be allowed to run in the browser.

However, to achieve full functionality, the user would also be required to have a mouse, microphone, and webcam. No other files, or software are required.

In the possibility that a desktop app is developed, then a small amount of space on the user’s drive will be required. The processing power required will also be minimal so there are very low system requirements:

* Dual Core Processor
* 1GB RAM
* An Operating System
* An internet connection

**Limitations of the Proposed Solution**

**Security**

There will most likely be security flaws in the final release of the app. This is because I do not have the knowledge required to keep it completely protected and so I can only do my best and ask for advice from others who have more experience in protecting websites from threats such as SQL injection and XSS attacks.

The server will not likely be at risk of attacks from hackers because the userbase will be small, so there will be little incentive.

**Servers**

The service will require a server to remain permanently online for the app to function correctly. This is a limitation because it will have running costs for whenever the app is available.

The server will be required to store account information for logging in and to store chats and route messages.

**Features**

The number of features will be always be behind other social media apps because I am the only person working on it. This is a problem because it would mean people would be less likely to switch from their current social media app to mine.

**Bugs**

Because there are so many different ways to interact with an app like this, it is likely that there will be some bugs in the final version. Although, this can be mitigated through rigorous testing.

**Success Criteria**

Here is a list of criteria that I would consider to be an indicator or how successful the project is.

1. **Simplicity**

Simplicity is a requirement for keeping the app friendly to new users as well as reducing the number of bugs and security vulnerabilities. One feature that will make the interface more simple is the use of icons and text for buttons so that the user understands what they are clicking on. To judge the simplicity of my solution I intent to take a survey of users who will give their opinion on whether the solution is complex, just right, or too simple. It will be successful if 60% of participants say it is just right.

1. **Modern User Interface**

A modern user interface is significant to maintain the app as a competitor to other social media apps which will encourage users to switch to the app. This is because some will not be so worried about privacy but instead just be looking for a smooth social media experience. To gage how successful the solution will be at meeting this success criteria, I will take a survey of users and ask them if they feel that the user interface is modern or not. The solution will have met this criteria if the majority of participants agree that it is modern.

1. **Messaging**

A core function of the app should be that users can send messages to each other over the internet.

1. **Privacy**

Users should be able to block other users and their data should be stored securely. Users should know how their data is being handled and feel that the app is being transparent.

1. **Hubs and Interests**

Users should be able to locate and add other users as friends for easier communication with them. They should also be able to join community hubs for chatting.

1. **Security**

User’s passwords are stored as hashed values in a database so that if they are stolen then they will still not have been compromised.

1. **Logging In**

Users can create an account and log in later on to be able to continue the same previous chats, calls and discussions from previous sessions.

**Design**

**Methods, Variables, Classes and Parameters**

|  |  |  |  |
| --- | --- | --- | --- |
| **Module:** | **Script/Method:** | **Variable Names and Types** | **Purpose** |
| Account Handling | databaseConnection.php | $db PDO object | Is used as the connection to the database and is assigned using the databaseConnection.php script. |
| $servername String | Is used when creating the $db PDO object to point to the location of where the database is |
| $username String | Is used to define which user profile should be used to log into the database with |
| $password String | Is the password used to log into the database profile |
| $dbname String | Specifies which database at the location that should be logged into |
| $charset String | Specifies the character set that should be used so that errors aren’t encountered when certain characters are used |
| loginRequestHandler.php | $email String | Will be set using a $\_GET for testing and $\_POST in the final version. Holds the email address that is entered by the user. This is then used to find the required account in the table. |
| $password String | Will be set using a $\_GET for testing and $\_POST in the final version. Holds the password that is entered into |
| $db PDO object | Is set by requiring “databaseConnection.php”. It acts as an object that allows database interaction. |
| $sql String | Is set as the SQL statement in the form of a string that will be used to locate the account in the accounts table. |
| $stmt SQL Operation | Holds the process that enacts the statement in $sql on $db |
| $userdata SQL Select Data | Holds the data that is fetched from the server by the execution of $stmt |
| password\_verify() PHP Function | A built in PHP function that checks a string that has not been hashed against a string that has been hashed to check if they match up. |
| registerRequestHandler.php | $username String | Set by $\_POST from user form and will hold the username that the user would like their account to be called. |
| $email String | Set by $\_POST from user form and will hold the email that the user would like their account to be associated with. |
| $password String | Set by $\_POST from user form and will hold the password that the user would like their account to be accessible with. |
| $repeatPassword String | Set by $\_POST from user form and will hold the second time the user enters the password to reduce the number of instances where a user accidentally makes an account with the wrong password and can’t access it. |
| $token  String | The string will be a 19 character randomised string that should be kept secure for each user. It can be used in combination with the username and tag to verify messages are being sent by the real user. |
| $tag  String | Used as a visual way for the users to send friend requests to specific users even if they share the same username as other people. Set randomly, but must be checked against the database for other usernames so that they do not share both a username and a pin. |
| $id String | Made using the date, time, username, and tag. It acts as a unique identifier that does not have to be kept secure and can be used for looking up information from the account database by other scripts. |

**Algorithms**

**Flowcharts**

Displaying The History of A Chat To A User

Diagram

Description automatically generated

Set number of messages can be a constant that is adjusted for the best performance but allows for users to scroll up in the chat for a short distance before more messages have to be gathered from the database.

This is a required function because it allows the user to see messages that they and others have previously sent in a channel so they can continue a conversation after closing and reopening a chat or read new messages that have been sent to them.

Allowing A User To Send A Chat

**Diagram

Description automatically generated**

This is required for the core chat functionality of the app which allows the user to update the chat data table with their message so that users on the receiving end will be able to display it on their screen. It also includes the requirement for how to notify users that are not looking at the channel that there is a new message waiting for them.

It is important to use this update system rather than a polling system because it is much more efficient. Otherwise users would essentially be either DDoS attacking the servers which would exacerbate with more users or leave user’s messages unchecked, increasing the time between when the sender sends the message and the receiver receives it.

Update chats on client

function checkAttributes(msg, updatedMsg)

newMsg = msg

for atr in attributes

if msg[atr] != updatedMsg[atr]

newMsg[atr] = updatedMsg[atr]

return newMsg

newChats = []

for msg in chats

newMsg = checkAttributes(msg, updatedChats[msg])

newChats.append(newMsg)

newChats.sort(DATE, NEWEST)

display(newChats)

Allowing A User To Log In

**Diagram

Description automatically generated**

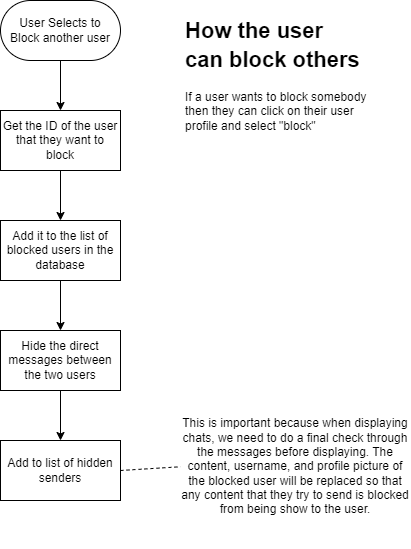
PHP has built in functions to test if emails are valid and to protect against SQL injection and HTML has attributes that ensure that fields are not left empty.

To achieve privacy and security, it is essential that it’s possible to differentiate between users. Therefore, the log in system is required, because it allows us to do this securely.

This also opens up possibilities for advanced personalisation and preference options that allow the user to customise the GUI, such as through “themes” that would change the colour scheme of the app.

Furthermore, users will already most likely be used to using social media apps that use similar log in systems, and so it would be extremely learnable.

**Blocking a User**

****

**Validation Methods**

General Inputs

All test inputs must be protected against SQL injection, otherwise this would be a massive security flaw. PHP has built in functions used to sanitise strings and validate inputs. To remove HTML tags, we can use the filter\_var($input,FILTER\_SANITIZE\_STRING) function which is important for displaying usernames and messages correctly. To protect the database operations, we can disable emulated prepared statements and use only real prepared statements. This should give an attacker no chance to inject malicious SQL. We can also introduce the data during the execution rather than while we prepare the SQL or prepare the operation. This ensures that the user’s inputted data is always treated as data, not instructions.

Email Input

Email inputs can be verified using filter\_var($email, FILTER\_VALIDATE\_EMAIL). I will not sanitize the inputted email, because then the email that is processed may be different than the intended email. Instead a warning can be thrown to the user about their input and the action can be cancelled. After this the user can amend the field.

Usernames

Usernames will be blocked from entering specific Unicode characters that may have connotations of racism, sexism, or other hate crimes.

Profile Pictures

Uploaded files must have an image file ending such as JPEG or PNG to prevent people uploading scripts. They should also be under 8MB in size. If they are larger than this then they can be compressed to see if they fit the size requirements after compression. If not, then an error can be thrown.

Passwords

Passwords should be longer than 6 characters and include numbers, at least one upper case and lower-case letter, and a special character (!”£$%^&\*()@?<>~#’`¬|-\_=+{}[]:;,.).

Pins

Pins should be 6 whole numbers between 0 and 9

**Pseudocode**

Friend Requests

userId = GET(userId)  
friendName = GET(friendName)

validate(userId, ID)

sanitise(friendName, INVALID\_UNICODE)  
  
check = db -> SELECT \* FROM account\_data WHERE username = friendName  
  
if check != null

db -> UPDATE account\_data SET incomingFR = userId WHERE username = friendName  
 return “Success”

updateUsers()

else

return “Error: No user with that username”

Deleting A Chat

msgId = GET(msgId)

senderId = GET(userID)  
  
check = db -> SELECT \* FROM messages WHERE sender = senderId, id = msgId

if check != null

db -> DELETE FROM messages WHERE id = msgId

updateUsers()

Updating Your Personal Description

desc = GET(description)  
tok = GET(token)

db -> UPDATE account\_data SET personalDescription = desc WHERE token = tok

updateView()

**Usability**

**Learnability**

It should be easy for users to transition to my app from other social media apps. Therefore a system based on simple familiar buttons and icons is required. After looking at the way large social media apps display their buttons, I have noticed every button is accompanied with a visual clue that it is a button when you hover over it. Usually this hover effect includes some pop up text explaining what the button does, a change of colour of the button, or a subtle change of shape.

Graphical user interface, table

Description automatically generated with medium confidenceGraphical user interface, application

Description automatically generatedGraphical user interface, text, application

Description automatically generatedGraphical user interface, text, application

Description automatically generatedText, chat or text message

Description automatically generatedGraphical user interface, application, icon

Description automatically generated

Another commonality is that there are vertical menu options at the sides of the screen. This makes sense because it uses up horizontal space where it is to spare on landscape screen viewing ratios, and thus it has become our go to place to search for menus.

As well as this, similar icons for certain functions are used across the web. By using these icons it mean that a user will immediately understand what a button does, allowing them to complete tasks quickly and easily.

Background pattern

Description automatically generated

**Efficiency**



Every action should be accessible within a few clicks of the home screen (as seen in the structure diagram below).

This will be done using a series of scrollable panes that span across the screen and use the aforementioned familiar icons. These menus will include the hubs menu, the channel module, the chat section, and the online user list. These will switch to reflect the hub that you are currently interacting with, or the direct message list that you are currently viewing. This will be easy to navigate for new or returning users because they can always return to the home page with the large, easy to locate home button in the top left corner of the screen.

**Errors and Validation**

Very few actions cannot be undone and so it’s difficult to make permanent errors. With each option will be the option to undo it by deleting the message or flicking a switch back or renaming your account etc.

If users enter invalid data then no action will be taken and the user will be asked to amend the invalid field. After they have amended the field then they can try and submit the data again, after which it will then be revalidated.

When creating an account, the user will have to enter the password twice to ensure that on the first entry, they did not make a mistake. This is relevant because the password will be hidden from the client screen as they enter it for security reasons. It’s also important that they enter the right password so that they can still log in when they want to.

If users accidentally insert problematic and invalid characters into a field, we can alert the user of the problem.

**Satisfaction**

The GUI will be designed around usability, aesthetics, and satisfaction. This will be achieved by using undistracting block colours for buttons and backgrounds which will give it the modern and futuristic look. When hovering over icons, they may subtly change shape or colour. Each action may also include a fun animation. Icons will be basic and will keep the same meaning throughout the app as to not cause any confusion. The page must also respond quickly, because a slow page can be frustrating.

**Structure of the App**

All options should be accessible within about 3 clicks of the home screen.

**A diagram of a home screen

Description automatically generated with medium confidence**

\*all movements through the menu can also be reverted to the previous menu page.

This shows that you will be able to complete actions based on our essential features very quickly which allows for fast paced use. It also makes it easier to understand how to complete certain tasks which reduced the time it takes to learn how a feature will work as well as gaining an intuition for how the app functions.

**Testing**

**Iterative Testing**

After developing each feature, I will check that it is working correctly before continuing on to the next.

**General Testing**

|  |  |  |  |
| --- | --- | --- | --- |
| Number of Testers | Method | Instructions | Explanation |
| 20 | Blackbox | Testers will be instructed to use the app how they normally would and explore the different available menus | This is to uncover any bugs that users may encounter when using the app normally. |
| 20 | Blackbox | Testers will be instructed to try and break and interact with the app as much as possible by spamming different inputs. | This is to find problems that may arise when a regular user is specifically trying to break the app using a brute force method. |
| 2 | Whitebox | Testers will be able to look through the code and try different actions on the site in order to try and break it. This includes entering invalid characters, formats and SQL code. | This is to reveal any bugs that may be very specific, or that could be targeted by attackers. |
| 1 | Iterative | I will test each feature after having developed it to make sure that it is working as intended. | This is important because when a user is using the app normally and as intended, then it should definitely not have bugs. |

**Specific Testing**

|  |  |  |  |
| --- | --- | --- | --- |
| Feature Being Tested | Method | Expected Output | Real Output |
| Account Registration | Whitebox.  The tester should first enter the registration screen and follow the on-screen steps to making an account. After that, they should repeat the test again. When making the account, they should use the inputs given below. The data from Login 2 should not be accepted so they should then proceed to Login 3 and end up with 2 successfully make accounts.  Data: Login 1  Username: “test\_user1”  Password: “Pa$$w0rd”  Pin: “123456”  Pfp Image: Any small image file  Login 2  Username: “<b>\bad\_test\_username</b>”  Password: “smol”  Pin: “dwed”  Pfp: Any small executable file  Login 3  Replace login 2 information -  Password: “Pa$$w0rd”  Pin: “654321”  Pfp: any small png | Checking the account\_data data table afterwards reveals that 2 new rows have been made, and the username and tag, token, and id are all unique. The first username should be as entered but the second username should have had the HTML tags and backslashes removed. The second password should be too small and therefore disallowed and the second pin should not be accepted.  The hashed passwords on the two different rows should be different if they have been salted. |  |
| Account Login | Blackbox  This test can be completed after the account registration test – it is essential that the data table contains only the result of this test when successful, and that all other entries have been removed. The user should enter the three sets of data (logging out between each) that they see below and see if it allows them to view an account and the message history.  Data  Set 1  Username: “test\_user1”  Password: “Pa$$w0rd”  Pin: “123456”  Set 2  Username: “not a real username”  Password: “Pa$$w0rd”  Pin: “123456”  Set 3  Username: “test\_user1”  Password: “Not a real password”  Pin: “324443” | They can view an account on the final set of data, but the tester is warned about their other inputs. |  |
| Friend Requests | Blackbox  Requires two testers. Both testers should be logged into separate accounts and should know the username and tag of each other. The first tester should navigate to the “add friends tab” and enter the information of the other tester into the required fields. They should then send the friend request. The second tester should then refresh their page.  The first tester should then enter the username “sBeve” and pin “3423”. | The second tester should see an option to accept or block the friend request and the first tester’s username and tag.  The second request should not come through and should just appear as a warning on the first tester’s screen that their inputs are invalid. |  |

**Large Data Set Testing**

This will be testing the databases of the server using large mock data sets of 1000 rows generated using “mockaroo.com”. The field type may refer to a data type option on the website.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Feature being tested | Randomly Generated Data Field Name | Field Type | Test Description | Expected Result |
| Logging in and Registering accounts | token | 20 character long random string | Insert the randomly generated mockaroo data and make another 10 accounts using any valid data. Try to log into the accounts and then pray |  |
| pin | Number between 00000 and 99999 (must have 5 digits) |
| password | SHA-1 Output |
| id | 20 digit integer |
| Username | Username |

**Databases**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Token | Email | Username | Pin | Hashed Password | PFP Token | pubKey | Friends | Friend Requests to | Friend Requests from | Channel Notifications | Channel Pings | Personal Description | Blocked Users |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |

**Account Data**

The account data table is used for storing information on users, most of which will need to be accessed during (blue), or right after (green) login. Other will need to be accessed as the user is interacting with the app.

After logging in, the information will be processed and automatically sent to the client to process the visuals. For example, the PFP (profile picture) token is a unique identifier used to locate the profile picture in a directory on the server. This image will then be sent to the client to be displayed at the top right of the screen.

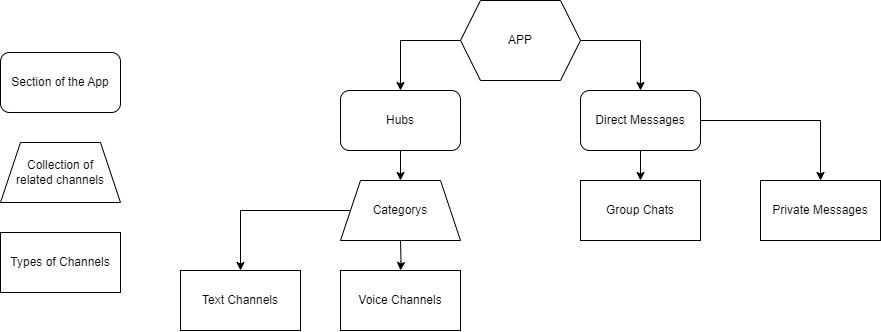
It is also essential that a client can access some information about other users, such as the PFP token, username, and personal description to help the user to identify exactly who they are speaking to.

**Messages**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ID | Time & Date Sent | Edited | Channel ID | Sender ID | Text Contents | Image Contents | Pinged Users | Verified | File Contents |
|  |  |  |  |  |  |  |  |  |  |

A database of messages is required, because some messages will need to be accessed by any user at a moments notice. This is because most channels in hubs will be public, meaning users will only need an invite, or for public hubs, a search, to be able to access the channel.

Message storage will be split into “channels”. A channel will be a link between users. It could be direct, only including two users that wish to talk privately, or it could be accessible by hundred of users that all have the same interests.



Key

**Hubs**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ID | Name | Number of Users | Date Created | Directory Location | Owner |
|  |  |  |  |  |  |

The information on how a hub should be organised and displayed needs to be kept as long as the hub is around. As hubs are supposed to be permanent social centres, the information must be stored keeping long term access requirements in mind. I have opted to store the information in both a specially made directory and data table row. The main server properties and attributes can be stored in the database (as seen above), and larger, more complex data can be stored in XML, image format, and JSON files in the directory.

Examples seen below:

Text

Description automatically generated

**Graphical user interface, text

Description automatically generated**

Text

Description automatically generated

**Graphical User Interface Designs**

**First Ideas**

This is my first idea about how the app may look. I’ve abstracted and labelled buttons.

Graphical user interface

Description automatically generated

I then made this to get an idea of what this would actually look like. I don’t really like the colour scheme right now but that can be finely adjusted when I am using CSS to tweak the look.



Finally I then removed the thick black borders to make it look more modern.



**Development**

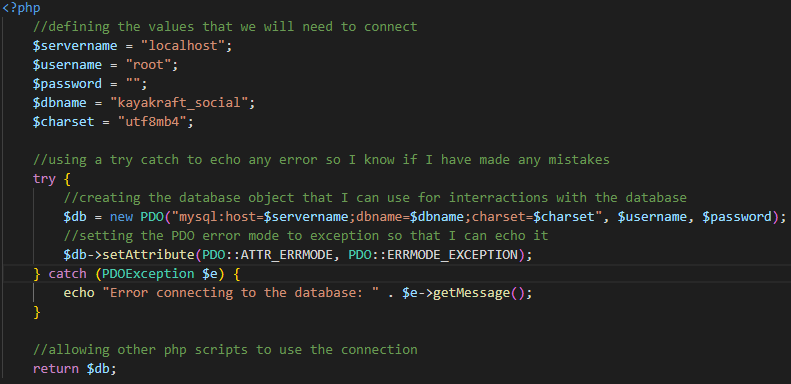
I started by thinking about what feature I must build first, and of course, to differentiate between users and therefore give us a base to build around. You can’t send a message or block somebody if there are no other users. This means that I had to start with an accounts management system.

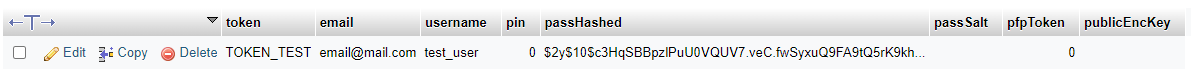
**Accounts & Friends Management System**

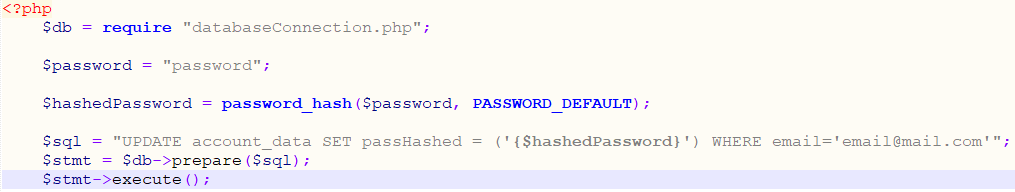
**Prototype 1**

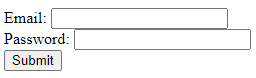
Base Ideas for Accounts and Database Interactions

For this, I would need to make a data table to store the account information. I decided to make the php script responsible for connecting to the database now so that I would be able to include it into other scripts later on.



Afterwards I made the “account\_data” table with a few of the necessary columns using phpMyAdmin.

I added the data for this user manually and set the password to “password” using the below script.

After this, I would then need to build the front and back end of the login page. I first made the front end using a basic HTML form. I decided not to add any CSS for now as it is a waste of time if I end up redesigning the web page in the future. 

A picture containing text, screenshot, software, font

Description automatically generated

(The comments above were ideas for how I might use javascript to send the form but this never comes to light).

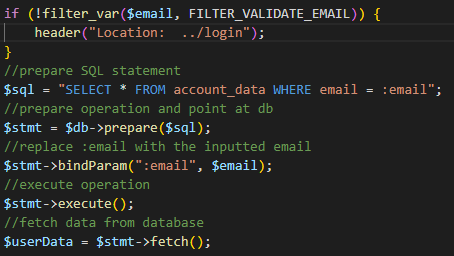
I then added some form validation in HTML as this was a simple and effective way of ensuring that the user wasn’t sending blank data to the server.



It was then time to build the back end which was pointed to by the html form.A picture containing text, screenshot

Description automatically generated

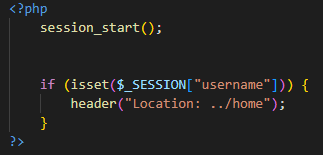
I annotated out the redirect to the home page so that I could then test the feature. I fixed some syntax errors that were throwing errors and soon it was working. When I entered the username and password that I had entered into the database, it would show “Success”. When I entered random or no values into the log in page, I would stay on the log in page even when submitting.

Then I had the realisation that this was not protected against SQL injection and discovered that I can use prepared statements to mitigate the risk. As well as this, I validated that the email was an email using an inbuilt PHP function. 

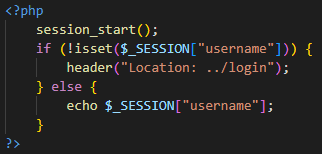
I wrote this down as something that should be tested later on.

After this, I decided to make the home page and make use of sessions and session variables to track individuals and show a unique home page so that I could tell it was working. I also decided to prevent users from accessing the home page if they were not logged in, or the login page if they were logged in.

Top of log in page:



Top of home page:

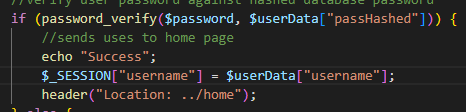


I tested it and it worked as intended, however, I learned that I was now stuck in one account, and could not log out. To fix this, I made a log out button on the home page.

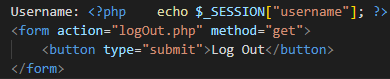
It points to this php script which destroyed the session and sent the user back to the login page.

A picture containing text, font, screenshot, line

Description automatically generated

The session system is currently using the username variable to check if a session exists. We should set this to our user’s username so we can show the user which account they are logged into. This is run when the user successfully logs in.

We can then display this in our home page.



Code Simplification

At this point I had a break in development for a couple of weeks, and when I came back, it took me a while to understand all my code and find where it was. This was a big problem because if I was struggling with just one feature, then how would another developer be able to maintain the code of the final solution? It was essential that I rewrote my code into a modular form. It was also only possible for users to log into existing accounts rather than creating their own to log into. This must be addressed.

**Prototype 2**

Simplifying and Modularising Database Operations

For Prototype 2, I need to restructure and plan so that I can keep my code organised. I decided to try the Model View Controller design pattern and use classes to simplify actions of my scripts. This would mean rewriting and improving my previous code.

I started with the database connection as I did before. Protecting it inside a class would ensure that all interactions with the database are internally validated, which would prevent me from accidentally writing code that would display errors to users.

A picture containing text, screenshot

Description automatically generated

This means that at the moment, we can only access the database directly from within this class. We should write methods for create, read, update, and destroy operations to streamline communicating with the database in the future.

Create:

A picture containing text, screenshot

Description automatically generated

I developed this for loop after researching methods of how you can include a variable number of parameters for a method. I decided that the easiest method would be using arrays. I decided on the for loop rather than a foreach loop to make it easier to assign the bound parameter to the right alias position.

Read:

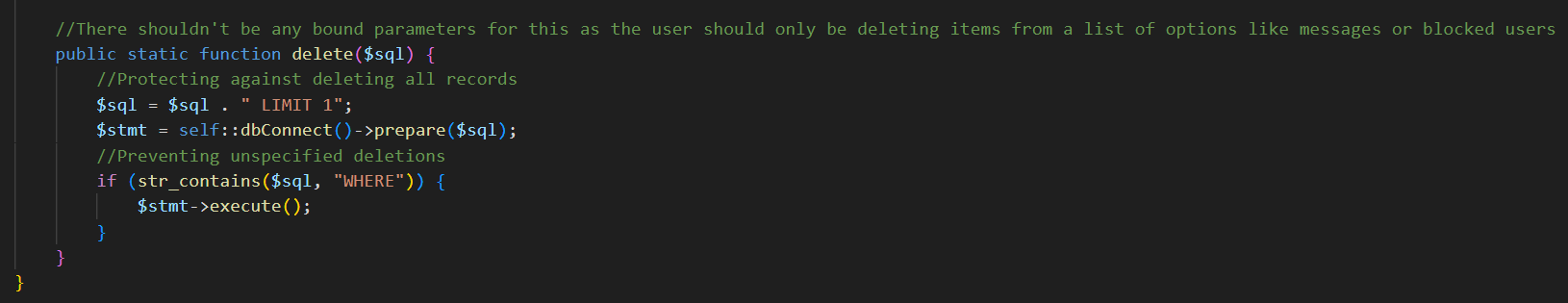
A picture containing text, screenshot, font

Description automatically generated

Update:A picture containing screenshot, text

Description automatically generated

Delete:



I have appended “LIMIT 1” to the Delete and Update SQL and checked for the key word “WHERE” because the update and delete SQL keywords will make changes to the whole database if they are not conditioned. This is absolutely not what we want.

I tested all of the functions that we had just created by creating a record, reading the data on it, then updating it, reading and echoing the new information, and then deleting it, and checking that it had been deleted on phpMyAdmin. I also tried entering invalid SQL statements into the update and delete functions, which did not pass. I also tried deleting two records and it was limited to 1. I was satisfied that everything was working correctly.

After this, I moved onto a log in system similar to what we designed previously, but updated to use the database class.

Log In System (Success Criteria 7)

Back End:

I wrote this PHP script that is pointed to by the HTML form when it is submitted. It will redirect the user to the home page and set a session variable so other scripts know which user the client is.

A picture containing text, screenshot, software

Description automatically generated

I then wrote out a similar login script to that of the previous Prototype, but with the database functions replaced with some new functions that I can then write in a new class shown below. I have also set the header locations with some added data to tell the front end to notify the user that some of their data is wrong. The file below is the accounts.php file that we have required into the script above.

A picture containing screenshot, text

Description automatically generated

This file contains a class which holds functions that are related to accounts management.

Now that the back end is complete, we can write the front end, and test.

Front End:

A picture containing text, screenshot, software

Description automatically generatedA picture containing text, screenshot, font, line

Description automatically generated

This is our front end HTML code. The main difference to our previous login page is that this now includes some PHP that will check for additional information in the header and update the style of the input boxes accordingly. After a few attempts, I came up with the code above, and tested.

Result for Incorrectly Formatted Email Inputted:

A screenshot of a login box

Description automatically generated with medium confidence

Result for Incorrectly Entered Email or Password:

A screenshot of a computer

Description automatically generated with low confidence

I will make this warning more easily understandable and look better later on, but for now this will do for testing.

I also then included the home page from the previous prototype to make sure that the login system was working correctly with sessions, and it was.

Registration System (Success Criteria 7)

Front End:

After this, I needed to make a registration page, and then finally run through my tests on the system before I can declare that it is all working as intended.

To start with, I added a button on the log in page which would allow you to make an account if you didn’t have one.

A picture containing font, screenshot, graphics, text

Description automatically generated

Which looks like this:

A picture containing text, font, screenshot, graphics

Description automatically generated

Then I made a web form in “register.php” and added the session redirector. I also made all the entries required and added some validation all inside a web form that would redirect to the registration attempt script.

A screen shot of a computer

Description automatically generated with low confidence

A screen shot of a computer

Description automatically generated with low confidence

Back End:

After this, I considered the validation and then wrote a new class called AccountFunctions, which handles functions that are run when handling accounts where no interactions with the database are required.

This went into the accounts.php file which I can use as a kind of library for any account manipulation.

A screenshot of a computer program

Description automatically generated with medium confidence

I only needed to write validation functions for the pin and password, because there is currently only one condition to be met for the username, and repeat password and email address and that can easily fit inside an if statement. I can easily add functions to validate these later with things like hate speech prevention and validating email addresses through sending an email and activating the account through clicking a link in the email.

After this I wrote a class that I can use to make account objects. A screen shot of a computer program

Description automatically generated with low confidence

I wrote getters and setters and a constructor function so I can add validation later if I need to.

Then I wrote a script that utilises these functions to create an account.

A screenshot of a computer program

Description automatically generated with medium confidence

It first gets all the values and validates them and then modifies the username to display correctly. Afterwards, it creates a new account object, sets the values and adds the account to the account\_data table. I then tested the code and found that it was all working as expected. However, there was no inclusion of setting the primary key in this so I added an ID generator.

The ID generator generates a random 19 digit number and this can then be included into our account functions with a few modifications.

I first added this into the registration attempt script.

A picture containing text, font, screenshot, graphics

Description automatically generated

This would then set the id using the following function.

A picture containing screenshot, text, font

Description automatically generated

To check that the id has not already been assigned, checkForId() is used.

A screen shot of a computer

Description automatically generated with medium confidence

I then tested all the code to make sure it was working properly. After fixing some syntax errors, it was all functional.

Conclusion of Prototype

There are still a few flaws in this design that I identified in testing and observed afterwards. Here they are along with why they should be fixed:

* Firstly, the values are validated outside of the Account class so any other time I want to use the class then I would need to validate the information again which adds unnecessary complication. (Keeping Code Simple)
* The IDs are generated using PHP, however, a system using SQL would be far quicker if it exists. The priority of this is minimal, because account creation is not a bulk task and the difference of optimising it would be negligible. (Improving app performance)
* Emails are not fully validated to make sure that they are not duplicated. They are also not validated to make sure that they are legitimate emails belonging to the user. (Improving security and functionality)
* The log in credentials to access the database are on show rather than being inside some kind of secure settings file. (Improving security)
* There is still no error catching, so if there is an error (such as if the MySQL server refuses to connect), it will display to the user rather than logging it or displaying to an administrator. (Better user experience and code quality)
* We are using many different session variables to store user information for the current client which means that all scripts that use them have to be updated to account for any new account attributes. (Keeping code simple)

I aim to fix these issues in the next prototype.

**Prototype 3**

Addressing Account Object Data Input Validation

To address the first issue, I will move the validation so that I can use the accounts class more freely in the future.

A computer screen shot of text

Description automatically generated

A screen shot of a computer code

Description automatically generated

This means I can now use the accounts class in the future, but there are still modules that I would like to add to it in the future for increased functionality and adaptability. I have built this so I can get the result of the validation by querying the result variables of the Accounts class.

I can then use the class and result variables to check and respond to any errors in the user’s input. To do this, I modified the registration script.

A screenshot of a computer program

Description automatically generated

I need to move the validation to the AccountFunctions class now, so that it can be used in the Account class.

A screenshot of a computer program

Description automatically generated

Here, I decided to use regex to validate the password input because it is very fast to process.

I realised that I could type the letter E into the pin input box, but scientific notation is not wanted for the pin so I made sure to flag them up. A screen shot of a computer screen

Description automatically generated

Ensuring Email Addresses are Unique

After this I moved on to validating that each user’s email was unique. First I updated the validateEmail function to include a check for other emails that are the same.

A black screen with white text

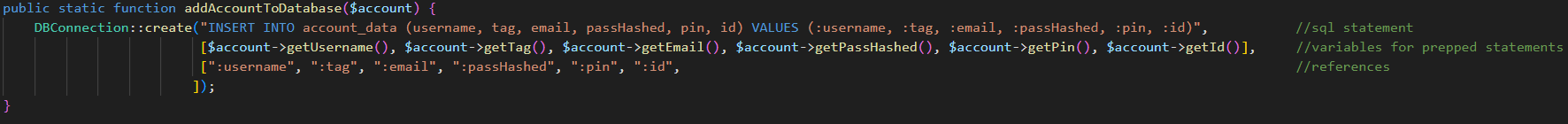
Description automatically generated

I then made this which will search the database for the parsed email.

A black screen with text

Description automatically generated

Now all we need to do is add the account to the database which is done by this function which takes our account object as a parameter and creates a new entry with all the inputted information.



This means that the registration process should be complete. We now have an account function that will nicely condense the session variables down into one object that can be used everywhere.

Using the Account Object for Session Identification

This means that the top section marked in red can be replaced by the last line.

A computer screen shot of a program

Description automatically generated

Success Criteria 7 – Being able to create and log into an account, has successfully been achieved.

**Prototype 4**

Storing Friend Requests and Friends in the Database (Success Criteria 5)

Now that users are able to register accounts and log into them, we need to give them a way of finding their friends accounts and saving them (befriending them). For this to work, we will need to store an account’s friends in a database so they can be linked together while the user is using the app.

I made 3 new columns in the accounts database as shown:

A close up of a sign

Description automatically generated

Incoming friend requests stores a list of IDs of users that are requesting to be friends with a specific user. Outgoing friend requests is friend requests that the user has sent themselves. Friends is a column to hold a list of friends that have been requested and accepted. I will be using the format of a JSON list so the default value for any account will always be [].

There is a flaw in this method of storing requests and friends IDs in that for each friendship or request made, two IDs are stored in the database. However, fixing this problem would mean a trade-off between data storage resources and computation resources that is not worth it. You would have to search through potentially the entire user base’s friend and requests to find if another person was your friend.

Making and Displaying Friends

Displaying Friend Requests:

I decided to make a page called “friend-manager.php” that is where you can manage all your friendships.   
This means you should be able to do the following:

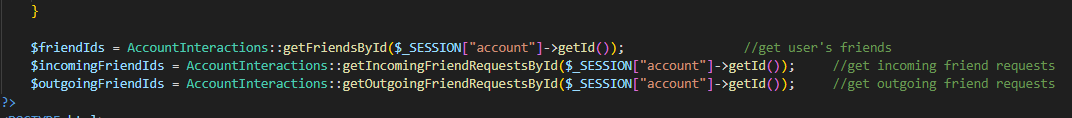
* Send friend requests
* Cancel friend requests that you have sent previously
* See any friend requests that have been sent to you and be able to accept/reject them
* See your list of current friendships and be able to remove them as a friend

To begin with, I made a title along with some unordered lists with titles explaining what will be contained inside them along with a home button and a friend request form.

A black background with colorful lines

Description automatically generated

To display all this information, it must be first gathered from the database.



This means that I must write some functions that will do this.

A screen shot of a computer program

Description automatically generated

A screenshot of a computer screen

Description automatically generated

I them wrote some embedded PHP into the friend-manager page to display this information as list elements with buttons that activate JavaScript functions.

A black screen with colorful text

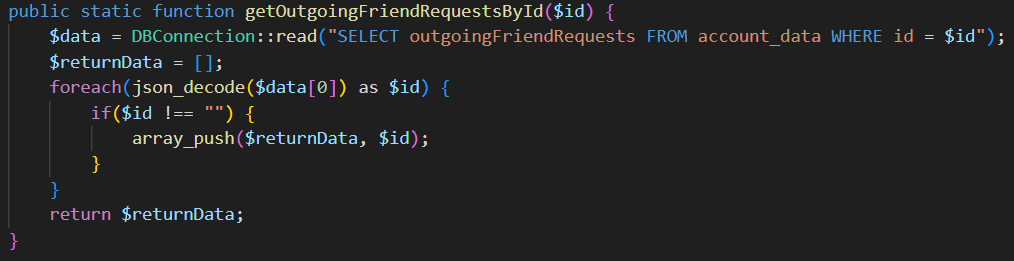
Description automatically generated

Then I repeated similar processes for the other lists.

A black screen with text

Description automatically generated

After some extensive testing a bug emerged where “[]” is sometimes decoded into a list with a single empty string item in it rather than an empty list. To fix this bug I added an extra check to remove any empty string items when lists are retrieved from the database.



Now all friends and friend requests will appear to the user in the friend-manager tab.

A screenshot of a computer

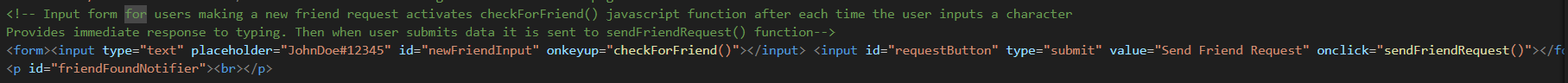
Description automatically generated

Sending and Accepting Friend Requests:

It’s essential that I must make it so that you can send and accept friend requests easily from the friend manager page.

I wanted the action of finding and interacting with your friends to be as easy as possible, so I decided to add the “friendFoundNotifier” paragraph element which will contain updates and information about actions that the user is taking on the friend manager page. This design was found to be intuitive and easy to understand.

I started with the form input that I made earlier with a text input.



Here you can see that when the user has just typed a character, it runs the function checkForFriend which you can see below.

A screenshot of a computer program

Description automatically generated

This gets the data that has been inputted using another function called getFriendInputData. It then sends an ajax request to the server to check for the account that has been inputted. If the account does exist then it will notify the user that the account they have typed in is real. To limit the number of requests querying the server, the requests are only sent when the correct format of username is inputted. This is validated by checkFriendInputValid.

A screen shot of a computer code

Description automatically generated

A screen shot of a computer

Description automatically generated

Then I made the back end which just needs to return the text to be shown based on the data inputted.

A screenshot of a computer program

Description automatically generated

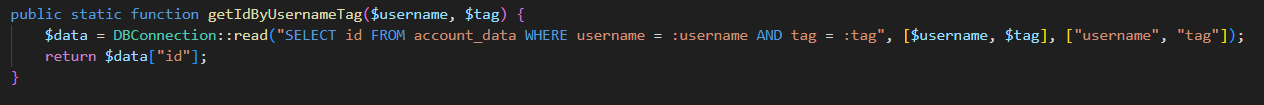
This uses some new functions that are defined in our accounts library.

checkForUsernameTag:

A black screen with a blue and orange line

Description automatically generated with medium confidence

getIdByUsernameTag:



checkForIdInOutgoingFriendRequestsList:

A screen shot of a computer

Description automatically generated

getOutgoingFriendRequestsById:

A screen shot of a computer code

Description automatically generated

checkForIdInFriendsList:

A screen shot of a computer screen

Description automatically generated

getFriendsById:

A screen shot of a computer code

Description automatically generated

checkForIdInIncomingFriendRequestList:

A screen shot of a computer

Description automatically generated

getIncomingFriendRequestsById:

A screen shot of a computer screen

Description automatically generated

Then here is a small test showing the result of the page with different inputs.

Existing User:

A close up of a number

Description automatically generated

Invalid format:

A close-up of a number

Description automatically generated

Incorrect Username:

A black and white rectangular object with black text

Description automatically generated

Incorrect Tag:

A close up of a number

Description automatically generated

Friend Request Already Sent:

A screenshot of a computer

Description automatically generated

Incoming Friend Request From User:

A screenshot of a computer

Description automatically generated

Already Friends:

A screenshot of a computer

Description automatically generated

Own Account:

A black and white rectangle with black text

Description automatically generated

Now that our user can enter a username and check that it’s valid. We need to actually send the friend request. When the send request button is clicked, it runs the sendFriendRequest function.

A screen shot of a computer

Description automatically generated

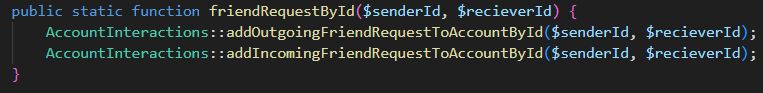
This simply tells the server that the user wants to be friends with the inputted user. This means our script that receives the information should add the two users as friends of each other if there is already an incoming fried request from the inputted user, or add a pending friend request if no request has already been made.

A screenshot of a computer program

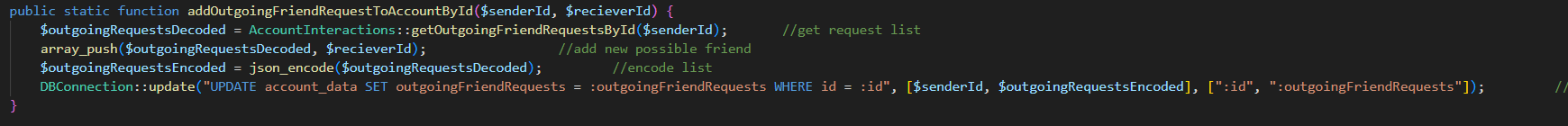
Description automatically generated

This uses new functions.

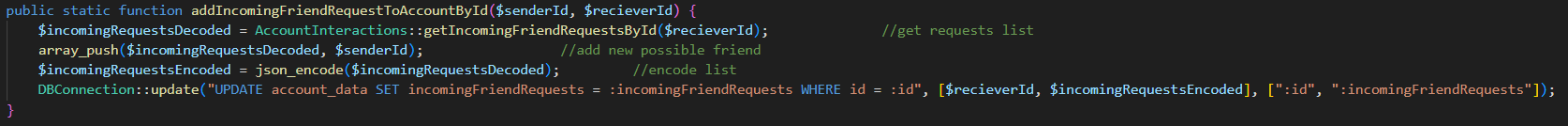
friendRequestById:



which uses addOutgoingFriendRequestToAccountById:



And addIncomingFriendRequestToAccountById:

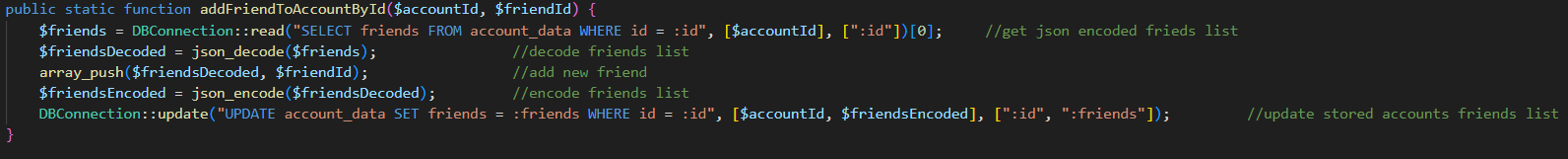


becomeFriendsById:

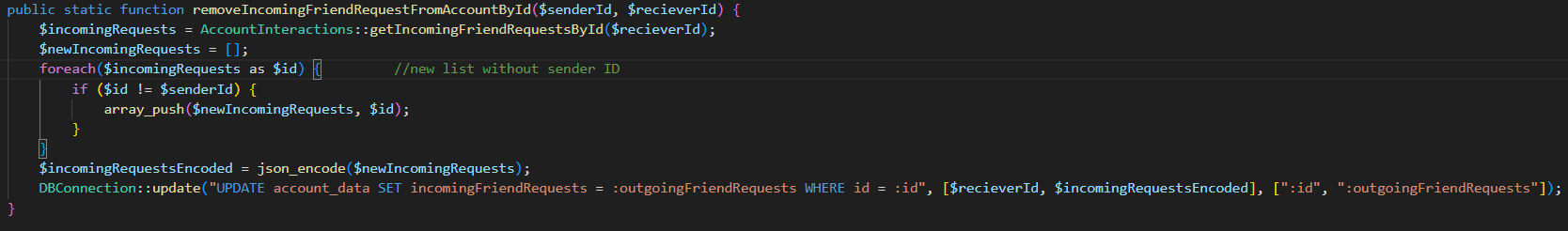
A screen shot of a computer screen

Description automatically generated

addFriendToAccountById:



removeIncomingFriendRequestFromAccountById:



Now this should allow us to send friend requests to others and it will appear on their friend manager page. Here’s a small test to show it working:

Before

Sender: A screenshot of a message

Description automatically generated

Receiver: A screenshot of a message

Description automatically generated

After

Sender: A screenshot of a message

Description automatically generated

Receiver: A screenshot of a message

Description automatically generated

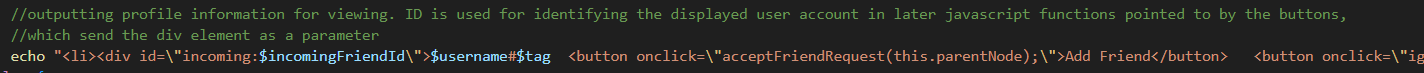
As you can see, this is working properly. Now what I want to do is add functionality to the three buttons that I have added. “Add friend” is an easy option to start with. This is because we already have the code for adding friends. It uses the same script as sending a friend request which will either add a friend (if a friend request is already coming from the other user) or send them a friend request (if no friend request exists). All the extra validation included in the script is useful for making sure the user doesn’t mess with the client interactions with the server to come up with any undesirable outcomes like a user befriending themselves or a user that they are already friends with.

Below, I connect the button to the script with an AJAX request similar to how I did before. This time I submitted the ID rather than the username and tag.

A screen shot of a computer program

Description automatically generated

The friend id is stored in the id of the div inside the list element as you can see below. This makes it easy to access as you can pass in the parent node of the button element and get the id as an attribute of that element.



Here’s a test to show that working:

Before:

A screenshot of a message

Description automatically generated

After:

A screenshot of a message

Description automatically generated

This is all working as intended. Then I worked on the cancellation of friend requests. I began by making a PHP script that can remove the request from the database. This calls the function defined in the database then responds to the client saying that it has worked.A black screen with colorful text

Description automatically generated

cancelFriendRequestById:

A screenshot of a computer screen

Description automatically generated

This just uses some of our previous functions.

After writing the script, I went on to call it from the client with and AJAX request.

A screen shot of a computer program

Description automatically generated

This uses the same technique as previously whereby the id of the friend is stored within the id of the div container. There is also a script that will display the message to the user that they have no outgoing friend requests. I added this in because it was not displaying properly when I was testing. Here you can see where the friend id is stored.



Here you can see the cancel button in action:

Before:

A screenshot of a computer screen

Description automatically generated

After:

A screenshot of a message

Description automatically generated

Now I will add function to the ignore button which is essentially just a cancel friend request button but for the receiver rather than the sender.

However, I decided not to reuse the old script so that I can easily verify that the user is only able to change friend interactions in relation to their account.

Here is the JavaScript function that is called by clicking the button.

A screen shot of a computer code

Description automatically generated

And here is the PHP script that is ran by the AJAX request. It uses cancelFriendRequestById as we have done before.

A screen shot of a computer code

Description automatically generated

Then I decided to allow users to remove friends. I achieved this similar to the other buttons on this page where I made an AJAX request to a PHP script. Here is the JavaScript function activated by the button.

A computer screen shot of text

Description automatically generated

And here is the PHP script that is calls.

A computer screen with text

Description automatically generated

This uses the removeFriendById function which destroys a friendship.

A screen shot of a computer

Description automatically generated

This uses the getFriendsById function which we have demonstrated before.

**Real Time Chat Service**

**Learning WebSockets**

My first approach was to find a PHP based WebSocket library that I could try to use. At first, I tried to use a library called Ratchet, but it turned out that although it claimed to work with windows, it was mainly focused towards Linux users. I also ran into numerous other issues and then discovered that it is really quite old as well.

After this, I decided to try to find a different programming language to build a WebSocket server with. I quickly discovered the possibilities using JavaScript with NodeJS, and thought it would fit the project well as it is fairly modern and quite powerful and would also be able to interact with the database. I planned to first get to grips with the WebSocket module (ws) for Node and the WebSocket API built directly into JavaScript so that I would then be able to later move onto using Socket.IO.

WebSocket API

I decided to make a very simple chat service for clients accessing the client.html file. First, I decided to write the server side, using the online documentation as guidance.

A screen shot of a computer program

Description automatically generatedFirst, I required the “http” and “ws” modules into the file.  
After this I made an http server to handle the http requests.  
Then I made WebSocket server piggybacking off the http server.  
After this, I added an event handler to display the request headers when a client tried to connect so I knew it was working.  
Then I added an event handler to the server for when a client connected and embedded an event handler to listen for incoming messages on the new WebSocket. I also logged it in the console to make sure the message info was coming through correctly.

After this I had to design the front end. To start with, I made an unordered list as a container for the messages and an input with a submit button calling a JavaScript function to take the message to be sent. Then I started on the functions and WebSocket handling.A screen shot of a computer program

Description automatically generated I got the html elements and assigned them to variables. I then established the WebSocket connection and bound it to the variable ws. After that I wrote a function to display a new message and set that so it would be triggered on the event that a message was received from the server. I wrote a function that would use the WebSocket to send a message back to the server when submitted.

Then I opened two browsers and opened the front end html file to test, and both were sending and receiving messages correctly.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

Two instances of WebSocket Headers were also outputted to the console as planned.

A computer screen shot of a program

Description automatically generated

Socket.io

After this success, I decided to move on to trying Socket.io. I wanted to do this because socket.io is a layer of the WebSocket API that adds a lot of built in features that make it much easier to work with. The main important one is automatic reconnections because the WebSocket connections can sometimes drop out.

To begin with, I decided to make an http server using the Express module for Node.js as the handler for http requests. I required in the Socket.io module and created a file named “public” in the same directory which the express server will serve up if you try to access the localhost on port 8000. I then created and bound the socket.io server to the express server and added an event handler to tell me whether a WebSocket connection had been made properly. I also added a disconnect event handler to log disconnects in the console. On connection, a “message” event is also emitted that I can attempt to detect on the client side.

A screenshot of a computer code

Description automatically generated

I created the following file in the public folder and called it basics.html. I added the Header text HELLO so that I knew I was actually accessing the file. I added the script with the source of the client side socket.io module that is made available in the public folder when the socket.io server is running. This makes the io object available. To check this worked, I tried logging the object to the console. Then I used the io function to connect to the websocket server on my localhost on port 8000 and created event handlers for connection and message events. Any messages should be logged and any connections should show the socket id.

A screen shot of a computer program

Description automatically generated

A computer screen shot of a program

Description automatically generatedA computer screen shot of a program

Description automatically generatedThis was the result in the browser JavaScript console. Here we can see the io object then the socket instance then the socket id (indicating that the connection has been made) then the message from the server that has been displayed. This means it is all working.

However, I was mainly interested in hosting a Socket.io server that I could access from any page of my current set up – none of which was being served up by the express server. From my previous testing with the WebSocketAPI, I knew I could get this to work using the built-in http module which is native to Node.JS. So I switched out the lines of code involving express and implemented an http server.

A computer screen shot of code

Description automatically generated

This time I decided to try to access it by setting up a connection from one of my web pages that was already accessible through my domain. At the start I was getting many errors claiming that my CORS policy was blocking the connection which I eventually fixed by accepting connections to the socket.io server from my domain. I also had to add socket.io to the client through a CDN service because previously, the client was including the client side version of socket.io that was being provided by the express server and Node. A screenshot of a computer program

Description automatically generated

I wanted to test this using a global chat that any users would be able to see on the home page so I coppied it over and modified it.

To do this, I just changed the function that is run when a message is received by the server to re-emit the message out to all clients as shown here.

A computer screen shot of text

Description automatically generated

I wrote the function showMessage to be run when the client receives a message event from the server. It starts by checking where the user is scrolled to. If they are within 20 pixel of the bottom of the chat box then they will automatically be scrolled down to see new messages when they are received. It then creates a new list element, fills it with the received data, and appeands it to the bottom of the chat box list.

A screenshot of a computer program

Description automatically generated

Here is the chat box and input field where messages are typed, as well as the button to send chats with.

A screen shot of a computer code

Description automatically generated

I applied this CSS to it to make it usable.

A screen shot of a computer program

Description automatically generated

I then tested it by running the server and logging in to two different accounts. I could then send messages between each of them.

A screenshot of a chat

Description automatically generatedA screenshot of a computer

Description automatically generated

Now that this is working, I can move on to integrating direct messages along with our friend system.

**Prototype 1**

Direct Messages

The direct messages are similar to the previous messaging that we created, except they have to be directed at a specific users.

Chat Server Socket Identities

Sending and receiving is still going to be done by the chat server, so first I need to identify users that connect to the server. The ID must therefore be stored on the user’s device sot that it can be sent to the chat server. I did this using a script that sends an AJAX request to the server and collects the ID whenever the page loads.



A computer screen with text on it

Description automatically generated

This uses cookies which I am accessing using two functions: setCookie and getCookie.

A computer screen with text on it

Description automatically generated

I will need to install a MySQL interaction module for Node to get the required information from the database. Here I create the connection with my MySQL database then log the connection and any errors in the console.

A computer screen shot of text

Description automatically generated

Then I created a listener for a new event called “id” which is where the client logs into the chat server using the logged in user’s ID. It starts by adding this to our clients list which is an array of JavaScript Objects that contain the socketId and the userId for each connection. It then gets thet username and tag from the database and logs it on the console. After this is logs the whole clients list to the console so you can see how many connections there are.

A computer screen shot of text

Description automatically generated

This is run on the client end which just sends the user’s ID as soon as the websocket connection is made. It uses the getCookie fucntion that I made earlier.

A black background with white text

Description automatically generated

Making Channels

Then I wanted to make chat channels. The channels will essentially acts as database entites that act as a participant list that a group of messages should display to in one place. If a users sends a message directed to a specific channel, then the chat server should get the participant list from the database and then send the message to each of the connected users. Here is what the table that I made looks like in the database.

A screenshot of a computer

Description automatically generated

I also need to be able to get all the DM channels that a specific user is a participant in, so I added the dms column

A screenshot of a computer

Description automatically generated

I wanted a direct message channel to be made whenever two users became friends, so I added this into the becomeFriends function.

A screen shot of a computer

Description automatically generated

This calls a function in a new library called ChatInteractions:

A screen shot of a computer code

Description automatically generated

Which in turn calls checkForDM:

A computer screen with colorful text

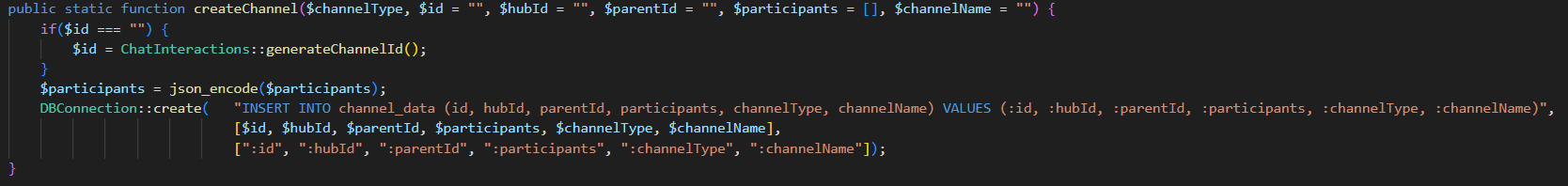
Description automatically generated

generateChannelId:

A black screen with colorful text

Description automatically generated

createChannel:



addDMToAccountById:

A computer screen shot of text

Description automatically generated

Now a channel will be made with both users when they add each other as friends.

Channel Selection

Now that we have channels, I need the user to be able to choose which channel they want to speak in (who they want to speak to) with different messages that they send. I decided to use the HTML select tag to display a list of friends that the user can select. I used embedded PHP to display the channels that the user can pick from.

**GUI Design**

I will be designing the GUI of the app using HTML and CSS and using native JS for functionality. Because all browsers come with their own default CSS, I had to reset all these changes before writing any of my own because if I didn’t, I would run the risk of the site looking different on different browsers.

Firstly, I started at the base column design and features that will be visible throughout most of the app. I will put this in a file called “global.css” and include it as a stylesheet on all pages.

To begin with, I made a container div for the page and made it a grid. Then I added panels for the different sections inside this grid.

A screenshot of a computer program

Description automatically generated

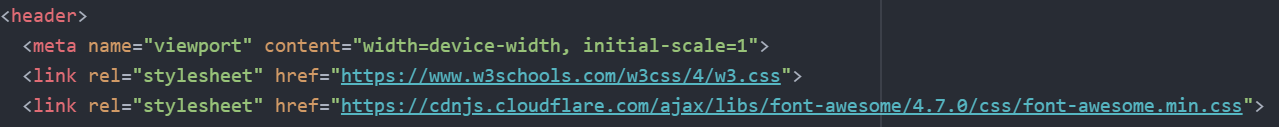
A screen shot of a computer

Description automatically generated

A grey rectangular object with white border

Description automatically generated

I then added a home button using an icon from the Font Awesome Icons Library and centred it in a div and a link button.



A screenshot of a computer program

Description automatically generated

A screen shot of a computer program

Description automatically generated

After this I experimented with placement of Hub icons on the hub bar. The hubs will eventually display their icons here as an identifier.

A group of circles with letters and numbers

Description automatically generated

A screen shot of a computer program

Description automatically generatedA screenshot of a computer program

Description automatically generated

My research into GUIs has previously indicated that there should be an indicator when you hover over a button, so I decided to add some little animations to the buttons.

A grey and blue button with black and blue buttons

Description automatically generated with medium confidence

Here, when you hover over the button, its colour changes to blue and the border radius decreases to make it appear closer to a square.

A screen shot of a computer program

Description automatically generated