

Web Technology Coursework 2 Report

Jonathan Binns

40311703

Introduction

The aim of this project is to create a coded messaging platform that uses the ciphers implemented in the previous coursework. This platform will have both server and client elements, with the server supporting CRUD (create, retrieve, update, delete) functions and allowing the persistence of data. My website should allow users to create accounts, log in via those accounts. It should display both a list of messages addressed to the user and the user should be able to select an individual message and have that displayed to them. The user should also be able to add messages which will be sent to another user. Finally, my server should allow the messages to persist along with the users. Ideally inside of a database, my server should also store data like passwords securely to make sure it adheres to basic security standards

Design

Below is the navigation diagram for my website. My plan is to use the sign up page as the 'homepage' by having it be the first page the user is taken to. Once on that page the user can either sign up for an account or click a button that takes them to the login page and they can log in from there if they already have an account. Once either the user is logged in or signed up they are taken to the home page, from there they can either go to a read message page or write message page. Both of these pages allow the user to go back to the home page. Design wise, im going to continue on from my aesthetic choices from the last project. My plan is to make the website as simple as possible to make it easy to use, there should be no unnecessary features as these would make the u.i. more confusing.

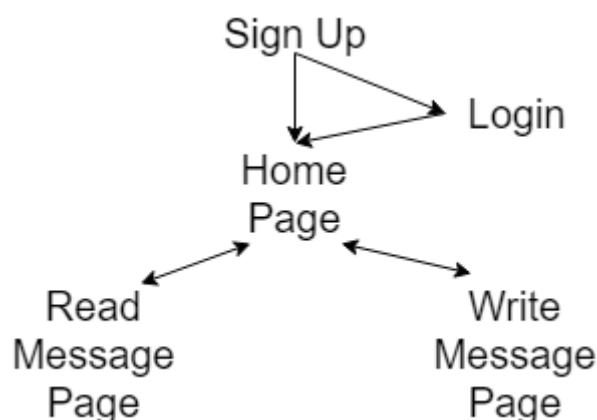


Figure 1: Navigation Diagram

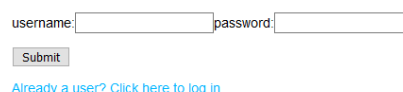
My signup page has one job, taking in the data for a new user and adding it to the database. It will do this by using some text boxes to allow the user to input their data

then make a post request to my user route, adding the user to the database. Once Done it will forward the user to the login page so they can access the home page. My login page is similar to my signup page in that it takes in data entered by the user and makes a post request, but unlike my signup page the post request will check that the data entered exists in the database and if true will render the home page. My home page will display a list of all the messages addressed to the user currently logged in. It will do this by making a get request to messages/<username> this route returns all the messages from the database when the recipient is the user that is logged in. The page will allow the user to click on a message and be taken to the individual messages page and will also have a link to the write message page. My read message page will show the contents of a message and allow the user to decrypt it. It uses the home page to take in the messageId and with that makes a post request to message/<messageId> this request sends them the decryption page and serves up the messages content ready to be decrypted. Finally, the write message page will allow the user to create a message. It will text boxes for the user to input all of the relevant fields to the message, it will then post to /messages and add the message to the database for it to be retrieved by the recipient. It will also allow the user to encrypt their message before sending with the click of a button.

Implementation

The most important part of my website's implementation is the app.js file. This file collates all the other separate files together and lets them work with each other. This file has mainly stayed stock from the express generation, but important parts have since been added. The most important of which is the portion of the code that creates the databases and initializes them with an entry each. This allows the database to be used by the other files and routes meaning that users and messages can be added and accessed. The app.js file also handles error routing throughout the entire website, this saves re-writing the same code over and over again and allows there to be a uniform look to any errors thrown by the website.

Sign Up

A screenshot of a web form titled "Sign Up". It contains two input fields: "username:" and "password:". Below these fields is a "Submit" button. At the bottom of the form, there is a link that says "Already a user? Click here to log in".

username: password:

[Already a user? Click here to log in](#)

Figure 2: Sign Up Page Screenshot

The first page the user will see when they visit is the signup page, this page allows a user to create a new account by entering a username and a password. Once these are entered they are submitted to the users route as a post request. This route takes in a piece of json containing a username and a password and adds it to the database. Once added the route then takes the user to the login page.

Log In

A screenshot of a web form titled "Log In". It contains two input fields: "username:" and "password:". Below these fields is a "Submit" button. At the bottom of the form, there is a link that says "New user? Click here to sign up".

username: password:

[New user? Click here to sign up](#)

Figure 3: Log In Page Screenshot

The login page is similar to the signup page as the user enters in their username and password but instead of making a post request to the users route, it makes a post request to the users/login route. This route then searches the database for the credentials that match the username that has been input and once found, then checks to see if they match the data input into the form. If they match the user is then taken to their homepage.

Home Page

[Add Message](#)

Enter messageID below to decipher

messageID:

```
[{"messageID":"abc","recipient":"ADMIN","content":"hello world","sender":"testname"}]
```

Figure 4: Homepage Screenshot

The next page the user will see after either signing up or logging in is their homepage. This page searches the database for each message where the user logged in is the recipient and then displays them. It calls the messages/<username> route to do this. Contained in this route is the SQL query to return all of the relevant messages from the database, it then parses these messages to the pug template and populates it with these messages. From this template the user can enter the message id to be taken to the decryption page.

Decrypt Messages Here

ciphred text: unciphred text:

[Decipher](#)

Figure 5: Read Message Screenshot

This page displays an individual message and allows the user to decrypt it. It calls the messages/<messageid> route which runs an SQL query to find the content of the individual message. Once found it calls the message template and populates it with the data returned from the query, the user can then click the decrypt button to show the true contents of the message.

Add Message

messageID:
recipient:
content (unciphred):
content (ciphred):
[Decipher](#)
sender:

Figure 6: Add Message Screenshot

Finally the write message page is used to allow a user to create a message. It has text boxes to take in all of the individual elements of the message and can encrypt them with the rail fence cipher. This calls a post request to the messages route which takes in all of the individual attributes of a message and adds them to the database. Once sent the form clears itself and lets the user either add another message or click back and go to the homepage.

Critical Evaluation

Overall, I think I have hit the most basic requirements set out. The server persists data for both the users and their messages allowing them to be viewed at any time. I think I achieved this aspect of the requirements as the server element does its job and stores data so the users can log in at different times and see all their messages no matter when they were sent. My website allows people to sign up and log in. To sign someone up my website takes in the data entered into that page then makes a post request to the users route to add them to the database. To log someone in it makes a post request to users/login, there it checks that the data entered is present in the database for that entry and if correct sends the user to their home page. These elements fit the requirements as they allow for users to be added to the database and checked, albeit in a very basic way. My add message page allows the user to encode and send a message. It takes in all of the relevant fields such as message id, recipient, content and sender. It then lets the user encode the content of their message with the rail fence cipher. Once the user clicks send the page makes a post request to the messages route which adds the message to the database, ready to be retrieved. This aspect of my website completes the requirement as it does its job in allowing messages to be sent to a specific user. My service also allows the user to access all of their messages at once, their home page displays a list of all the messages on the database where they are the recipient. Once the user sees a message they would like to decode they input the message id to a text box and then are taken to the message page. The message page is very simple in that its job is to decode a single message so that the user can see the content. It makes a get request to message/view to display the messages data then the user can click on a button to decipher the message.

The first improvement I would make to my website is adding validation. The validation so far only checks if the user is valid if they go through either a signup or a login route whereas if they just went to messages/<username> they would be presented with a list of the messages addressed to <username> no matter if they knew the password for that account. I would implement validation using a cookie that is unique to each account and on entry to messages/<username> that cookie is checked to see if it is the correct one for that account. If it is then the page is shown, otherwise an error is shown. If I had more time I would also include password hashing and a password recovery route to allow users to regain access to their accounts if they forget a password. The next thing I would improve is the u.i. of the website. I would make it look a lot more polished and better spaced out. Overall I would just take the elements that are already there and space them out to make the website look better. I would also have each message in the home page be a link to its respective message page so the user could decipher it a lot easier. I would also spend more time to make the URL routes more user friendly. I would change things so that the home page could be just '/messages' and allow users to navigate between pages a lot better. Finally I would add more fields to user profiles to make them more full featured. I would add things like email addresses to allow the users to reset their passwords, I would also add things like bio's and profile photos so users could have a higher degree of personalization when it came to their profiles.

Personal Evaluation

This coursework has given me an understanding of all of the different elements that make up a website, especially when it comes to the server side. This was shown to me when learning node and express, which helped me to really see what goes on behind the scenes at my favorite websites. Node also helped me further my JavaScript knowledge as I had never used it for an application like this before. The second thing this coursework taught me was the how server requests work, before I knew that a client would have to make a request to a server but I had no idea about the different types of requests and what they can do. When writing my get and post requests I learned a lot about how websites transmit data. Finally the last thing I learned was the concept of templates when it comes to web pages. Using pug taught me that there are specific languages to deal with filling up a page with content from the server rather than relying on HTML. The biggest challenge I faced was knowing where to put my methods. I knew how to do all of the individual aspects, such as sending and receiving a message, but learning that you would have to put the add message method on a post request which would take in the message as JSON and the view message on a get request were concepts that I needed to learn and see for myself. The next challenge I faced was how these requests relate to the websites URL. My idea to have the home page of the website be user /messages/<username> meant that I had to keep the username of the logged in user as a variable so I could generate the new URL for adding a page. Finally the last challenge I faced was how to link the server and client sides of the website. After a lot of research I eventually came to the conclusion that they should be rendered on get requests meaning that the data returned from that request can be directly entered into the template. It also means that all pages to do with one database can be kept together with all of the requests.

References

Mardan A. (2014). Pro Express.js, Master Express.js: The Node.js Framework For Your Web Development. Used for: Understanding response methods and requests

Academind. (2017). Building a RESTful API with Node.js. Accessed on: 19.04.19
https://www.youtube.com/watch?v=0oXYLzuucwE&list=PL55RiY5tL51q4D-B63KBnygU6opNPfK_q Used for: understanding routes and basic get/post routes

Sqlitetutorial. (2019). SQLite Node.js. Accessed on: 19.04.19
<http://www.sqlitetutorial.net/sqlite-nodejs/> Used for: understanding SQLite syntax being in line with JS

Stack Overflow. (2017). Pug (Jade) HTML form. Accessed on: 19.04.19
<https://stackoverflow.com/questions/43090770/pug-jade-html-form> Used for: creating the form to signup and log in