Project Handover Documentation

Getting Started Guide and Overview of Project

By Glasgow University 3rd Year Project Team SH14

Contents

Introduction

As well as this document, there is also a getting started video available at: https://drive.google.com/file/d/1h-jElLlyz_OyLh6QONTXA2SheoYJId1b/view?usp=sharing.

This is the handover documentation produced by the members of Glasgow University Team Project Team SH14 for GoodBridge, a webapp that allows like-minded freelancers to match based on their moral values, interests, and location. The release accompanied by this documentation implements the GoodFriends functionality which allows matching between freelancers. It has not fully implemented GoodWorks, which would match up those looking to hire freelancers with those looking for work.

The following technologies are used in the codebase:

Database: MongoDB

Backend: Node.js, Express.js, Mongoose

Frontend: React.js, TailwindCSS

See the acknowledgements at the end of this document for a full list of node project manager packages used in the codebase.

Installation and Running Locally

Here are instructions for running the code on one's own computer. This allows you to access the website and its database without having to use an external server. The video guide explains all these steps, excluding the environment variable setup, as we aim to include these files in the handover code. If these two .env files are not present, they will need to be created.

Installing NodeJS

- Go to the official Node.js website at https://nodejs.org/en/download/.
- Download the appropriate installer for your operating system.
- Run the installer and follow the prompts to install Node.js on your computer.
- After the installation is complete, open a terminal (MacOS/Unix) or command prompt (Windows) and type node -v to verify that Node.js is installed and to check the version number.

Setting up the Codebase

- (Windows only) Unzip the '*sh14-main*' folder.
- Open the '*sh14-main*' folder.
- In the '*frontend-react*' folder create a file called *.env* (make sure to include the full stop at the start) and paste the following text into it:

```
REACT_APP_BACKEND_URL="http://localhost:3003"
```

- Then, in the '*backend*' folder, create a new file called *.env* and paste this code into it:

```
DB_USERNAME="goodbridgedatabase"

DB_PASSWORD="SOQqwqeajdX0NdqN"

BACKEND_URL="http://localhost:3003"

FRONTEND_URL="http://localhost:3000"
```

```
JWT_SECRET="goodbridgethissureissecretive"
```

Installing Dependencies

- In a terminal navigate to sh14-main/frontend-react
  - Type the command 'npm install' and hit enter
- Navigate to sh14-main/backend
  - Type the command 'npm install' and hit enter

Running the Server Locally

- Open two terminals.
- In the first terminal change the folder using the command "cd /fronted-react" and run the command "npm start".
- Change to the second terminal and change the folder using the command "cd /backend" and run the command "npm start"

Getting Started

Logging in and signing up

- In a web browser, go to http://localhost:3000/
- You should see the homepage for Goodbridge, with two buttons 'login' and 'signup'
- Signup will take you to a form with three fields.
  - Name must not include special characters except single underscores and length must be between 2 to 50 characters.
  - Email must be of a valid format.
  - Password must be strong, with capitals, numbers, special characters and of length >=8
- Pressing submit will attempt a signup and will let the user into the main site if successful, otherwise will alert the user of the errors in their input.

ReactJS Frontend Components

Overview of Components

The frontend is built using ReactJS and was initialised using Create React App. Styling is done with TailwindCSS.

The React components are all stored in frontend-react/src/components. App.jsx and index.js are outside this directory and are in frontend-react/src. The components directory contains three subdirectories: /images, /pages and /utils.

- The *images* directory contains the logos and backgrounds used in the project.
- The *pages* directory contains two more directories /goodFriends and /goodWork, and the page components.
  a. *goodFriends* contains the landing page, and match page components for GoodFriends functionality
  b. *goodWork* contains the same for as above, but for goodWork. This section is mostly incomplete.
- The *utils* directory contains miscellaneous reusable components, such as NavBar.jsx and dropDown.jsx

Component Descriptions

App.jsx & Index.js

App.jsx

The App component is used to specify the URLs used in the webapp.

It contains two instances of the built-in <*Routes*> component. The first <*Routes*> component contains pages which should only be accessible to authenticated users. It is enclosed within a *PrivateComponent*, which limits access to said authenticated users only. The second <*Routes*> component is public and contains pages which are accessible to everyone.

Additionally, the App component includes two functions: *whichGood* and *useEffect*.

- The *whichGood* function checks the browser's current URL to decide which background image to use.
- The *useEffect* function makes a request to the backend server and redirects the user to the "Resendemail" page if the response is false.

There is also a *PrivateNavBar* component specified here, which is only visible to authenticated users.

index.js

This renders the App component in strict mode. This is a development tool which adds additional checks during rendering, and should potentially be removed or replaced in the final production version of the product.

Pages

This section of the report will specify all the pages included in the codebase, which are written as .jsx files. These files can all be found in the *frontend-react/pages* directory.

Public Pages

These are pages available to the public upon opening the page, before logging in.

About.jsx

This page returns HTML for the about page, which specifies some information about GoodBridge.

Contactpage.jsx

This returns HTML of the contact us page. It provides various contact options for the company. It also makes use of react-icons for the Facebook, LinkedIn, and Twitter logos.

Homepage.jsx

This is the landing page which users will see when first accessing GoodBridge. It is a sleek welcome page which invites users to sign in or sign up.

News.jsx

The news page, which currently displays a list of articles. Each article includes a clickable link to the original source. Ideally, in the future this would be updated so the articles could be easily updated without prior knowledge of the code. Sanity.io may be a useful tool for this.

SignUp.jsx

The sign-up page. The component allows users to sign up for the application by entering their name, email, and password.

Once the user submits their information, the data is sent to the backend server using the *collectData* function. If the sign-up is successful, the user data and authentication token are stored in the local

Commented [IH1]: Clarify

Commented [IH2R1]: Ask Max

Commented [IH3R1]: Checks if the user is authenticated

Commented [IH4R1]: If not back to resend email

storage and the user is redirected to the email verification page. The redirection functionality is provided by the *useNavigate* hook from the React Router library.

Login.jsx

The login page, which a form to take the user's email and password and send them to a backend server using the *handleLogin* function. If the login is successful, the user data and authentication token are stored in the local storage and the user is redirected to the landing page. If the login fails, an alert message is displayed.

The form also includes links to the forgot password and sign-up pages. The redirection functionality is provided by the *useNavigate* hook from the React Router library.

GoodFriends Pages

This section contains pages which are only for use with the GoodFriends functionality. They are found in the *frontend-react/pages/goodFriends* directory.

landingPage.jsx

The landing page which users are brought to after logging in/signing up. React component fetches data from a backend server and displays a title and a button to start the matching process. The matching process is started automatically using the *useEffect* hook and the fetched data is stored in the result variable.

Commented [IH5]: Does the button on this page actually do anything now?

Commented [IH6R5]: Just links to the page. Automatically the matches happen

Commented [IH7R5]: Just to get the data, if not goes back to resend email

matchesPage.jsx

The matches page where users are able to see all of their current matches, and details about them.

React fetches the matched user's data from a backend server and stores it in the matches state using the *useState* hook. The *getMatch* function is responsible for fetching the data from the backend server.

The matched users data is passed to the MatchedCards component as a prop. The MatchedCards component is responsible for rendering each matched user card. The *useEffect* hook is used to fetch the matched user data once on page load.

Commented [IH8]: Is this depreciated?

Commented [IH9R8]: No, it calls the other bit

matchPage.jsx

The matching page, where users can see potential matches and either accept or decline the other user as a GoodFriend.

React fetches the potential match user's data from a backend server and stores it in the matches state using the *useState* hook. The *getMatch* function is responsible for fetching the data from the backend server. The potential matches' data are passed to the MatchCards component as a prop. The MatchCards component is responsible for rendering each potential match user card. The useEffect hook is used to fetch the potential match user data once on page load.

Commented [IH10]: As above re. decpreciation

GoodWorks Pages

There is only one page for this section, a coming soon page. The *frontend-react/pages/goodWork* directory contains this.

comingSoon.jsx

The coming soon page, which displays a message informing the user that GoodWorks has not yet been developed.

Private Pages

The following are pages which should only be visible to logged in users.

ProfilePage.jsx

The Profile page. React component that displays the user's profile information and includes a Delete Account button. The ProfileCard component is responsible for displaying the user's profile data and the DeleteAccountButton component is responsible for handling the deletion of the user's account.

ResendEmail.jsx

The resend Email page, where users are prompted to verify their email address. They are also able to resend a new verification email in case they didn't receive the first one. The component first retrieves the user's email from the local storage and sends a GET request to the backend to trigger the email sending process. If the email is sent successfully, the user is redirected to the GoodFriends landing page. If the user did not receive the email, they can click a button to resend the email, which sends a POST request to the backend.

forgotPasswordSend.jsx

This page provides a form that allows the user to enter their email address and send a request to the backend server to reset their password. When the user clicks the "Send Email" button, a POST request is made to the server with the email address provided. If the request is successful, the server will send an email to the user with instructions on how to reset their password.

forgotPasswordChange.jsx

After having made a request to change their password, this is where users can enter and confirm their new password. The component extracts the token from the URL parameters and sends it along with the new password (and re-entered confirmation password) to the backend server using the *changePassword* function. If the passwords do not match, an alert is displayed. If the password change is successful, the user is redirected to the login page.

Utils

These are the various utility components in use in the file, which are used throughout the rest of the site.

dropDown.jsx

This is a custom dropdown component implemented using React. It renders a button that, when clicked, displays a dropdown menu with options that can be selected. This have been used to display drop down menus on the left- and right-hand sides of the hidden nav bar which appears when logged in. The dropdown also includes a feature that closes the menu when the user clicks outside of it.

Commented [IH11]: What

Commented [IH12R11]: DropDown is the sides, big is the middle

bigDropDown.jsx

This is the drop-down component used for the central, horizontal drop down from the middle of the hidden nav bar component. It is used to access matching and matches pages.

completeProfileCard.jsx

This is used as a base point from which users can update their details. It contains three modal buttons, each of which allows the user to update their personal details, select values, and choose jobs of interest. The component uses useState and useEffect hooks to retrieve the user's existing values and jobs of interest from the backend server. The values and jobs data are passed as props to the corresponding sub-components, ChooseValues and ChooseJob, respectively. The sub-component PersonalDetails allows the user to update their personal information.

ChooseJob.jsx

This displays a list of job categories, allowing the user to select up to 4 jobs and their related skills. It fetches the data from a backend API and updates the user's profile with the selected jobs and skills. It

also keeps track of the user's selections using useState hooks and updates the UI accordingly. Once the user has made their selections, they can submit their choices, and the data is sent to the backend for processing.

ChooseValues.jsx

This allows users to choose values from a predefined list and limits their selection to 5 values. It includes a click handler function to toggle the selected state of each value, and a button to submit the user's selection to the backend server. The component stores the selected values in its state and updates them whenever the user makes a change.

PersonalDetails.jsx

This renders a form for users to update their personal details. The component uses useState, useEffect, useFormik, and react-select hooks for state management and form handling. The form allows the user to select their work type, hourly pay rate, and location using select and react-select elements. When the user submits the form, the collectData function is called, which collects the form data and sends it to the backend server for storage.

CustomButton.jsx

This is allows us to make the buttons used in the homepage. It takes two props: "text" and "action". The component renders a button element with the given text as its label, and the "action" prop as the function to be called when the button is clicked.

deleteAccountButton.jsx

This is the button which allows users to delete their account. When the button is clicked, the user is asked to confirm their decision. If the user clicks "Yes", their account is deleted, and they are redirected to the homepage. If the user clicks "No", the modal disappears, and the user is returned to the previous screen.

modalButton.jsx

This is used to create the pop up menus which appear for selecting new values, choosing jobs/skills, and updating one's profile details.

jobs.js

This is an array of job categories with their respective skills and values. Each job category has a unique ID, a name, and an array of values that describe the required skills or attributes for that particular job. The job categories range from administration to writing, and the skills required vary from technical proficiency to interpersonal skills.

logoutButton.jsx

This is used to log the user out and is accessed from the right hand side drop down menu of the hidden nav bar.

matchCards.jsx

This is the display used to show new unmatched accounts to the user. The component uses useState and useEffect hooks to manage the state of the matches displayed and the index of the current match being shown. It also includes functions for making "yes" and "no" choices on each match and communicating those choices to a backend API using fetch requests. The component uses conditional rendering to show the current match and animate the transition between matches when a choice is made.

matchedCards.jsx

This is similar to the matchCards, but for users which we have matched with. It renders a card for each match displaying the person's name, their shared values, and a button to connect with them via email. The email is pre-populated with a default subject and message.

hiddenNavBar.jsx

This is the navigation bar which appears at the top of the page when the user is logged in. It displays the logo of either "Good Works" or "Good Friends" based on the current path, and includes links to various pages, such as the home page, match page, and profile page. It also includes a dropdown menu for settings and user information, which includes a profile card and a logout button.

NavBar.jsx

This is the navigation bar used before the user has been logged in.

PrivateNavBar.jsx

This is used to decide whether to render the NavBar, or HiddenNavBar components.

PrivateComponent.jsx

This is used to prevent authenticated-user only components from being visible to those who are not logged into GoodBridge.

profileCard.jsx

This is used as a root from which the rest of the user's profile is displayed, it uses 2 subcomponents: profileSmallCard and updateProfileCard.

profileSmallCard.jsx

This is used currently just to display the user's name, although future installations may wish to expand upon this.

updateProfileCard.jsx

This allows the user to update account details, such as their email address and password.

The ExpressJS Server Backend

Overview of server

The server is currently run on localhost, however the environment variable 'BACKEND_URL' can be changed to a link to a deployed server.

Overview of routes

The routes that comprise the backend are divided in three major groups, account, auth and match. These are specified in *backend/routes.*

Account

Account routes deal with functions and features related to updating values of a particular account. The following are the routes that comprise this group:

- ***deleteAccount***: search for the account in the database and deletes it.
- ***getBio***: When a request is received, it first verifies the token passed with the request using the verifyToken function from backend/utils/token. If the token is valid, it fetches the user with the specified id from the database using User.findOne(). If the user exists, it sends the user's bio property as a response. Otherwise, it returns an empty response.
- ***getJobs***: When a request is received, it first verifies the token passed with the request using the verifyToken function. If the token is valid, it fetches the user with the specified id from the database using User.findOne(). If the user exists, it sends the user's jobs property as a response. Otherwise, it returns an empty response. If an error occurs during the process, it logs the error to the console and sends a 500 Internal Server Error response.
- ***getPersonalDetails***: When a request is received, it first verifies the token passed with the request using the verifyToken function. If the token is valid, it fetches the user with the specified id from the database using User.findOne(). If the user exists, it sends the user's location property as a response in JSON format with a 200 status code. Otherwise, it returns an empty response.
- ***getValues***: When a request is received, it first verifies the token passed with the request using the verifyToken function. If the token is valid, it fetches the user with the specified id from the database using User.findOne(). It then sends the user's values property as a response. If the user does not exist, it returns an empty response.
- ***newEmail***: When a request is received, it first verifies the token passed with the request using the verifyToken function. If the token is valid, it checks if the specified new email exists in the database by searching for a user with that email using User.findOne(). If a user with that email exists, it sends a response indicating that the email already exists. Otherwise, it sends a response indicating that the email is available.
- ***setBio***: Sets a Bio to the user using the site, to do this it uses the MongoDB updateOne command, selecting the appropriate user ID and using $set to insert the bio value obtained from the front end into the database.
- ***setJobSkill***: Extracts the skills and user properties from the request body. It processes the skills array to group the skills by job and create a jobSkills object. It then updates the jobs property of the user with the specified _id using User.updateOne() with the jobSkills object. If the update is successful, it sends the update result as a response. Otherwise, it sends a 500 Internal Server Error response.
- ***setPersonalDetails***: Like the other set routes, it updates the user's document to reflect the changes selected in the front end. In this case the data inserted using the User.updateOne() command and the $set instruction is the location of the user.

Commented [IH13]: Just updating the bio field

Commented [IH14]: The method to MongoDB which updates just one user

- ***setValue***: This route uses updateOne and $set to insert the values that the user selected in the value selection screen into the database.
- ***updateEmail***: When a request is received, it first verifies the token passed with the request using the verifyToken function. It then checks if the request body contains a newEmail property and if it is a valid email using the validator.isEmail() function. If either condition is not met, it sends a 400 Bad Request response with an appropriate error message. If both conditions are met, it updates the email of the user using User.findOneAndUpdate() and sends the updated user object as a response.
- ***updatePassword***: Checks if the request body contains a newPassword property and if it meets the required password strength criteria using the validator.isStrongPassword() function. If either condition is not met, it sends a 400 Bad Request response with an appropriate error message. If both conditions are met, it hashes and salts the new password using the hashNSalt() function and updates the user's password using User.updateOne(). It then sends the updated user object as a response.

Auth

Auth (catch-all term for authentication and authorisation) relates to every route tasked with ensuring users have the proper authorization to access the different areas of the webpage. The following are the routes in this group:

- ***confirmation***: The confirmation route takes in a jwt token (signed with the user's email address) from the URL and decodes is using the jwt secret key. Once the email address has been decoded, it finds the account attached to said email address and sets its verification status to true (effectively verifying the account). This route is used when the person clicks on the link that was sent to their email address. It takes them to this backend page and that will be followed by their token in the URL.

- ***login***: This is used for the login process. It depends upon JWT and Express technologies. It firstly checks the database for an instance of the input email. If this passes it then checks the input password matches the actual password using middleware in backend/utils/password.js. If this is successful, it signs the JWT token and returns the user and token as a response, allowing login.
- ***signup***: This route involves: validating the input fields such as name, email, and password; checking if the email already exists in the database; creating a new user; and sending a confirmation email. The password is hashed and salted before storing it in the database. If any validation checks fail, the router returns an HTTP 400 Bad Request status code along with a JSON object containing an error message. If the signup is successful, the router returns a JSON object containing the user data and the JWT token.
- ***resendEmail.route.get***: This route simply returns the verification status of a user's account to the frontend. This GET request is sent in '*App.jsx*' and '*ResendEmail.jsx*'. if the request returns false, App.jsx will return the user to ResendEmail.jsx, and if the get request returns true and the user is on the resend email page (ResendEmail.jsx) they will be automatically navigated to the landing page.
- ***resendEmail.route.post***: This route receives an email address from the frontend and passes it to the sendConfirmationEmail() function. This function then creates and sends a confirmation email (complete with a link) to the specified email address, using that email

address to sign a jwt token. This route is used on the '*ResendEmail.jsx*' page. When the resend email button is clicked, there is a POST to this route.

Match

This directory contains routes that enable the implementation of the matching procedure, which is a key component of the website. It may help to refer to the Matching Algorithm section of this document for clarification here. The routes associated are the following:

- ***getMatched***: This route sends the information related to the users' matches to the front end so that it can be displayed. For clarification, this relates to users that have successfully matched. It sends the following data of each matched user: name, email, values, and ID.
- ***getPotentialMatches***: Similar to getMatched, instead getting the users from the potentialMatches array and sending the following information to the front end: name, email, values, ID, bio, location. This information is the used by the match page to display the potential matches a user have, giving the power to the user to either say yes or no to the match.
- ***goodFriends***: This route handles the primary matching functionality of the website. When called, it executes the matching algorithm and then uses updateOne and $set to insert the result of the matching algorithm into the users document in the database.
- ***makeChoiceNo***: This occurs when a user declines a potential match. In this case the algorithm first checks to see if the other user has already added us to their awaitingMatches array by saying "Yes" to us and if that is the case, it removes us from their awaiting list. Additionally, it removes the other user's ID from our potentialMatches array, effectively completing the flow and rejecting that user. If the other user hasn't sayed "Yes" to us yet, it just removes their ID from our potentialMatches array.
- ***makeChoiceYes***: This occurs if a user has said yes to a potential match. The algorithm checks to see if the other account has already said "Yes" to our initial user. To validate this, the awaitingApproval Array of user 2 is traversed trying to find the first Users's ID. If it is found, then a match is made, and both users are moved to the matches Array, which can then be displayed in the matches screen. If the initial user is not in the awaitingApproval array of user number two, then User two's ID gets moved to the initial user's awaitingApproval array.

Backend Utils

This section will explain the files in the *backend/utils* directory.

- ***emailConf.js***: This util has one function, sendConfirmationEmail(address). This function simply creates a nodemailer transporter (described below) and uses that to send a confirmation email to the specified address. The confirmation link address is constructed using a jwt token signed with the specified address.
- ***emailPass.js***: This util has one function, sendPasswordEmail(address). This function simply creates a nodemailer transporter and uses that to email a reset-password link to the specified email address. The reset-password link address is constructed using a jwt token signed with the specified address.
- ***goodFriendsMatch.js***: This is described in matching algorithm.
- password: See middleware crypto.
- token: See middleware jwt.

Middleware Used

Nodemailer

Nodemailer is a is a module for Node.js applications which allows easy email sending. Nodemailer is used in two specific parts of the code: to send the user confirmation email and the reset password email. It works by first creating a transporter object and then assigning it the email and password of the address you want to send the email from. Then upon sending the email you can assign it a subject, a body, and a recipient. Nodemailer can be tricky to use when using more traditional email services (such as Gmail), as these services have security measures to stop applications like nodemailer from accessing them. Therefore, if using something such as Gmail, it is advisable to use an “app specific” password, so that the nodemailer transported will have no problem with authentication. Another thing of note, would be that certain email providers will have precautions to stop “spam” emails from being sent, keep this in mind if sending bulk emails every day as you may get locked out of the account.

JWT

Json Web Token (JWT) is used to ensure that the user is who they say they are. The token is signed/created on login or signup and lasts 24 hours. In every interaction with the backend, the token is verified. The verification is in *backend/utils/token.js.*

Crypto

This library is used to encrypt the password before being stored in the database. In backend/utils/password.js the function *hashNSalt*() encrypts the string passed to it. *VerifyPassword*() decrypts the hash, salt and iterations and checks it is the same as the string password.

Outline of Matching Algorithm

The following is a detailed explanation about how the matching algorithm used in GoodBridge works. This section covers our translation to JavaScript of the original code (Python) given to us by our customer. It is different in some areas to the original code, for example, the original included an email dissemination functionality and was made for reading data from Excel sheets, functionalities that are replaced in our application. Nevertheless, the core logic of the actual matching has remained intact.

Most of what is explained here relates to GoodFriends matching, since it was agreed with the customer that shifting our full attention to polishing that functionality was the better option over attempting to produce sub-par algorithms for both GoodFriends and GoodWorks.

Sub-Functions

The matching algorithm is made up of one main function with multiple sub-functions that allow it to work. For it to be clearer we will start by defining and explaining the sub-functions, so that understanding the main function is easier. These functions are found in the file goodFriendsMatch.js.

- *matchValues(user1Values, user2Values, noValues)*: This function takes the values of each user and the minimum number of values needed for a connection. The two users' values are compared to each other to find the intersection between the two sets (i.e., how many values both users share). If the intersection's size is => the specified minimum number of matched values, the function returns an array of the form [true, matchingPoints] which will be used in the main function. The functionality of matchingPoints is explained below. If the minimum is not reached, then an array of the form [false, 0] is returned.
- *matchcountry*(user1Location, user2Location): this function takes as arguments the location field of each user. Location is an array of the form [country, city] so for this function the system checks if the first value of each users' location array is the same. If it is true is returned, if not then false.
- *matchcity*(user1Location, user2Location): This function works exactly the same as *matchcountry*, with the only difference being that it accesses the second value of the location array [country, city]. If the value is the same, it returns true, else it returns false.
    - It should be noted that currently we match by location, however the customer had previously expressed a preference for matching by time zone, and ideally this would replace pure-location matching in future releases.

Main GoodFriends Matching Function

The main function is *GoodFriendsMatching*(user1, database, no_values).

It takes for parameters the current user that is using the webpage, the database of all users expressed in an array, and the number of minimum values required (no_values) for a match (currently always 1). We obtain both the values and location of user 1 and save the values for them to be compared to the rest of the database. An empty array potentialMatches is created, where the results of the algorithm will be stored.

After this has been done the matching process begins. This is implemented by a iterating through each user in the database. At the start of each loop the variable matching_score is set to 0 and the location and values of the i^{th} user (database[i]) are saved.

First, matchvalues (see above) is ran, and if the result is of the form [true, matchingPoints] then the amount contained in matchingPoints (an integer) will be added to the matching_score variable. If the result is of the form [false,0] nothing happens.

Then, matchcountry and matchcity are called, in that order. They are extremely similar, so the procedure is the same. If the result of these functions is true, 10 points are added to the matching score. That means that if two users share country and city, that is 20 points added to the score.

Finally, the algorithm checks if matching_score is above the current threshold of 30 points. This threshold could be changed in the future if too many/few matches are being made.

If the threshold is met, the users ID is added to the potentialMatches array. This array then contains all users considered similar enough to the user which we are finding potential matches for.

This covers the full functionality of the GoodFriends algorithm, whose goal is to identify the potential users that a particular user might be interested in. After obtaining this group, it's up to the user to either accept or reject the match.

MongoDB NoSQL Database

Overview and Database schema:

The database that supports the GoodBridge webpage and its functionalities is hosted in MongoDB, a NoSQL database which is straightforward to use. For the project, the most important schema/table is that of the User. User.js is the backbone of GoodBridge and includes the necessary fields to allow for proper GoodFriends functionality. It also contains fields which could be used in the future GoodWorks feature.

Depending on the implementation of GoodWorks, more fields might have to be added such as a separate potentialMatches, awaitingMatches and matches array. The fields which are used by GoodFriends, and their respective datatypes, are as follows:

- name (String): The username of the account
- email (String): the email associated to the user.
- bio (String): A description of the user to be able to express more free-range thoughts not constricted by pre-determined values.
- password (Object): The user's password, stored securely (hashed and salted). The object contains three items, the hash(str), the salt(str) and the iterations(int).
- verifiedAccount: Boolean that affirms if the account has been verified during the signup process.
- age (Number): The age of the user
- location (Array): An array with 2 values; index [0] collects the country, [1] collects the city
- potentialMatches (Array): a list of users which the matching algorithm has identified as being suitably similar to the current user.
- awaitingApproval (Array): a list of users who have said they would like to match with the current user.
- matches (Array): a list of all confirmed matches.
- visited (Array): a list of all users the current user has 'seen' (said yes or no to).

The following fields are in the schema, but are not currently in use for matching as they are for GoodWorks:

- timezone (String): the user's current time zone.
- timezoneDifference (Number): the max amount of time zone difference a user would tolerate between those who they are matched with.

- locationPreference (String): signifies which type of work does the user expects to be offered. There are three types, in-person, remote and hybrid.
- pay (Number): states how what hourly rate the user expects to be paid, so job offers can be filtered.
- -hiring (Boolean)
- -freelancer (Boolean)
- -jobOffer(Array)
- -jobs (Map of Array): This map structure has for key the type of job, while the value is an array that holds the respective skills related to that job type that the user possesses.

Collections and documents:

Our MongoDB database is contained in a collection, which is the term used for the group of documents that form the database. Documents in this case are the representation of GoodBridge users, which are created and updated according to the User schema outlined above. During development, our collection has been limited in size, but nevertheless we feel the current structure provides a strong foundation for future growth. The tool we used for accessing the collection and check the different documents is MongoDB Compass, an official application that allows for intuitive navigation of the database. From here it was possible to see the values that each user has in their respective fields, while also supporting the possibility to manually edit and update the documents as needed.

Notes on Security

We have not fully implemented security provisions across the site. However, we have developed some degree of security on authentication and authorisation.

Authentication and Authorisation

- Uses JWT for authorisation and has a login/signup barrier with email authentication required before access to the site is granted.
- Password are encrypted using crypto library and sha256 hashing.

Known Other Vulnerabilities

- There is no XSS prevention, and this will need to be completed for deployment.

Testing

We used two type of tests that work with react apps which are Unit tests and Snapshot tests.

Unit Tests

Unit tests are used to test individual components of the React application. They verify that each component renders correctly, has the correct behaviour, and reacts appropriately to user interactions.

Snapshot Testing

Snapshot testing is a technique that takes a snapshot of a component's output and compares it to a stored version of the component. If the output of the component changes, the snapshot test will fail, indicating that something has changed that needs to be investigated.

Licensing

The project is licensed on a 5 year exclusive-use royalty-free license. This allows for the codebase to be used as desired by GoodBridge Ltd over the duration of the royalty-free period.

Acknowledgements

Authors

- George Yarr 2553638y@student.gla.ac.uk
- Andrei Mihai Stoica 2546896s@student.gla.ac.uk
- Ieuan Harries 2549037h@student.gla.ac.uk
- Andres La Riva Perez 2564664l@student.gla.ac.uk
- Max Cattanach 2555886c@student.gla.ac.uk
- Turki Almutairi 2614579a@student.gla.ac.uk
- Hao Xu 2431074x@student.gla.ac.uk

Credits to third-party libraries and tools

List of NPM packages used in the frontend:

├── @material-tailwind/react@1.2.4

├── @react-icons/all-files@4.1.0

├── @testing-library/jest-dom@5.16.5

├── @testing-library/react@14.0.0

├── @testing-library/user-event@13.5.0

├── autoprefixer@10.4.13

├── country-state-city@1.0.5

├── formik@2.2.9

├── framer-motion@7.6.7

├── jsonwebtoken@9.0.0

├── postcss@8.4.19

├── react-dom@18.2.0

├── react-icons@4.6.0

├── react-router-dom@6.6.1

├── react-scripts@5.0.1

├── react-select@5.7.0

├── react-simple-typewriter@5.0.1

├── react@18.2.0

├── tailwindcss@3.2.3

├── validator@13.9.0

└── web-vitals@2.1.4

List of NPM packages used in the backend:

```
├── cors@2.8.5
├── crypto@1.0.1
├── denv@1.0.4
├── dotenv@16.0.3
├── express@4.18.2
├── jsonwebtoken@9.0.0
├── mongodb@4.13.0
├── mongoose@6.8.3
├── nodemailer@6.9.1
├── nodemon@2.0.20
└── validator@13.7.0
```