NHS Breast Screening Awareness App Documentation + User Manual

Introduction

This manual aims to explain both the usage and the structure and design of our system. Each section of the document contains a brief explanation of the functionality and purpose of that part of the system, this is then followed by a technical description to aid system administrators working with the software.

The source code for this project is available on GitHub at
https://github.com/szmcook/NHS-Breast-Screening-App

The project can be viewed online here: https://nhs-breast-screening-survey.herokuapp.com/

Setup/Deployment Instructions

This system is composed of 4 core parts that need to be set up and deployed before use.
To complete the set up you will need:

- A Heroku account (or another web hosting alternative)
- A GovNotify API account (or another SMS sending API)
- A Google Analytics account
- A MongoDB database (or similar)
- To follow the instructions below we recommend you have a GitHub account

Setting up the database

In order to store and read data from the database the system must be connected to a database. The system uses MongoDB to host the database with Mongoose to connect to the database. The following steps describe how to set up and connect our system to a MongoDB database.

To read more about the details of the database please refer to the section titled Database Structure and Usage or to read more about MongoDB please refer to their website.

1. Create an account:
    a. Set your organisation name and project name to whatever is suitable
    b. Choose JavaScript as your preferred language
2. Create a shared cluster (the free option)
    a. Choose your cloud provider and region (in testing AWS on the Ireland server was used)
    b. Choose your cluster tier and additional settings (in testing these were left as default)
    c. Set the name of your cluster
    d. Wait for the cluster to be created
3. Navigate to your cluster sandbox and click the “collections” button
4. Create a database with the name “NHSResponseData” and collection name “Responses”
5. Set up your IP Access list:
    a. In the security tab select “Network Access” and then “Add IP Address”
    b. Create a new IP address entry in the Access List (in testing access was unrestricted however when deployed this should be set to the IP address of the server the web app is hosted on)
6. Create a new Database User
    a. In the security tab select “Database Access” and then “Add New Database User”
    b. Use Password authentication
    c. Set the name and password of the new user
    d. Give the user “Read and write to any database” privileges or higher
7. Connect to the database

a. In the cluster sandbox, click connect
b. Choose a connection method using Node.js 3.6 or later
c. In the project code, open the .env file in the server folder
d. Change the MONGODB_URI constant in the to the link as instructed by the MongoDB dashboard

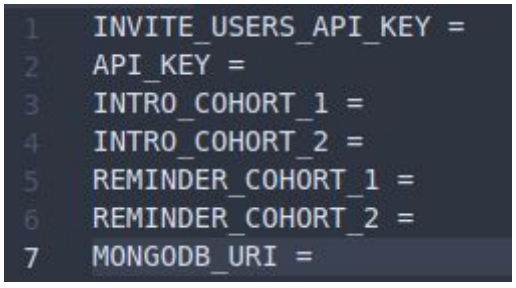
The system should now connect to your MongoDB database.

Setting up GovNotify

The server uses a script named 'notify.js' which handles the sending of SMS messages to users. By interacting with the GovNotify API and the input JSON object, the program sends personalized messages to a user. To read more about GovNotify please refer to the section titled SMS Messaging with GovNotify or to the GovNotify documentation.

To set up your GovNotify account to be able to send messages to users one should follow the steps outlined below:

1. Obtain a 'Live API' key from GovNotify by navigating to the 'API Integration' > 'API Keys' > 'Create an API Key' and create a 'Live' key. The API key must be copied into the .env file , one must take care with this process as once the API key has been viewed it cannot be viewed again.
2. Create all appropriately named template keys and in the .env file:



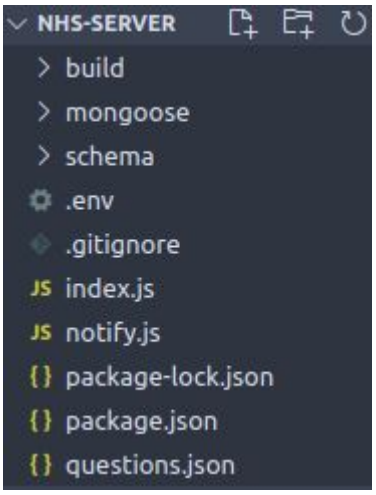
a. INVITE_USERS_API_KEY: the API key used for the inviteUsers endpoint
b. API_KEY: the GovNotify API key
c. INTRO_COHORT_1: intro text for the users on the family journey
d. REMINDER_COHORT_1: reminder text for the users on the family journey
e. INTRO_COHORT_2: intro text for users on the participants journey
f. REMINDER_COHORT_2: reminder text for the users on the participants journey
g. MONGODB_URI: the generated link to the MongoDB database

The .env file should include (but is not restricted to) these variables assigned to their appropriate API and template ID keys. It is necessary that the exact same variable names are used in the .env file.

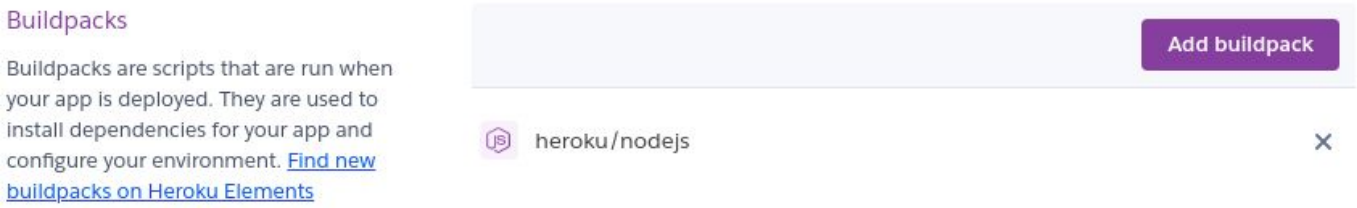
Deploying the web app

Deploying the web app can be thought of in four sections. The first step is to download the source code and make any edits desired. The second step is to build the frontend so that the server is able to serve it. The third step is to publish the server code and the frontend build to GitHub so that it can be accessed by Heroku and the final step is to create an Heroku web app and deploy the code. The instructions below cover these steps in more detail.

1. To start the process, clone the GitHub repository to obtain the codebase for the system.
2. Configure the .env file according to the instructions in the sections on setting up GovNotify and MongoDB.
3. If you want to make changes to the code, follow the information in this documentation in order to do so.
4. In order to comply with Heroku's folder structure requirements it's necessary to build the frontend locally. This is done by running the command $ npm run build from within the client directory.
5. You then need to copy the 'build' folder created across to the server folder. At this stage the server folder contains all of the code required to deploy the web app. It should have a structure similar to the one shown below.



6. To deploy the application to Heroku it is recommended that you follow the following steps. This is not the only way to host the web app but it has been tested and proven to work.
7. Upload all of the code (excluding the node_modules folder) to a GitHub repository.
8. Create an Heroku web app at Heroku.com (you will need to sign in/register) and give it a name.
9. Connect the web app to the GitHub repository containing the server and the pre-built frontend in the Deploy tab of the application dashboard.
10. In settings/Buildpacks make sure that the buildpack Heroku/nodejs is selected.



11. In the deploy tab of the Heroku dashboard press Deploy. There are various other setting such as automatic deploys that you may wish to enable.

Setting up Google Analytics

Setting up Google Analytics can take one of two forms, the first is a transferral of admin rights for an existing Google Analytics Property, and the second form is to set up Analytics monitoring from scratch.

To read more about our use of Google Analytics please refer to the section titled Google Analytics or to Google Analytics own website.

1. To transfer the admin rights for a GA Property that is already configured, simply navigate to the admin page of the property's dashboard and select "Property User Management". This page allows for user permissions to be added and modified. Click the plus symbol and "Add users",

then enter the email address (which must be linked to a Google Account) and select the relevant permissions. Click “Add”.

2. Setting up Google Analytics from scratch is more complicated and requires access to the frontend source code (accessible on GitHub).
    a. An Analytics account is required. This can be an existing account or can be created using an existing Google Account.
    b. Create an Analytics Property. There is extensive documentation for this process available from Google. Please note that the property should be set up as a “Universal Analytics Property” rather than the default “Google Analytics 4 Property”.
    c. Once set up, the property will be given a unique Tracking ID. This Identifier string needs to be added to the project to link the web app to Google Analytics. Please copy and paste the tracking ID into the Google Analytics initialization line within the useEffect function of “App.tsx”, in the code for the frontend. An example using Tracking ID “UA-191723436-1” is shown below.

```
// ON PAGE LOAD:
  useEffect(() => {
    ReactGA.initialize('UA-191723436-1');
```

    d. The frontend should then be built and deployed as explained in “Deploying the web app” above.

User Roles/Paths

Our system is designed to serve two distinct user journeys, throughout this documentation these are referred to as the participant journey and the family journey. The participant journey is for women aged between 50 and 71 who can expect to be invited for breast screening themselves, the family journey is for people with relatives in the participant group. The usage of the app and experience is the same for each user journey - the difference is in the set of questions that are asked.

The set of questions asked can also vary within the two groups of users, a user’s answers to one question may affect the set of questions that will be asked later in the survey and our system is able to accommodate this.

Question Architecture Design

Our flexible question architecture allows the survey app to implement several powerful features, including response specific branching and an adaptive progress bar.

In order to add questions to the survey, the questions.json file will need to be modified. This file contains a list of questions, to which new questions can be added for seamless integration, on the condition that format outlined below is observed.

The questions are stored in a JSON object which has fields for the index of the initial question on the family user journey, the index of the initial question on the participant user journey and an array, called Qs, which stores the questions.

The questions are represented as individual JSON objects in the array Qs, meaning they can be accessed by their index. The structure of the individual question objects is described below.

Item	Explanation	Format/Example?

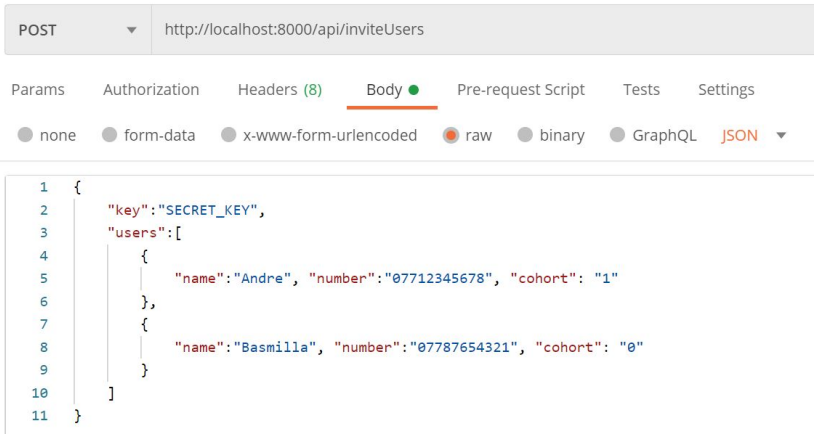
id	Each question has a unique id used to identify it within the system.	`"id":"q1Family"`
text	Each question has the main text of the question.	This is stored as a String
options	Each question has a set of options that a user can pick to answer the question.	These are stored as an array of strings. `"options": ["Yes", "No", "Not Sure"]`
nextQuestionIndex	The next question to be asked is dependent on the answer to the current question, therefore each question stores all the possible questions that could come next. These each correspond to one of the response options (see above). If the index for the next question is '-1', this signals that there are no further questions following the current one.	These are stored in an array of integers which are used as indices for the array of questions. `"nextQuestionIndex": [1,-1,-1]`
longestPath	Each question stores the length of the longest possible path through the survey from this point. This information is used to accurately calculate the user's progress through the survey and update the progress bar.	This is stored as an integer.
info	Each question also stores the information that should be displayed to a user once they've answered the question. As the information to be displayed is dependent on the user's answer to the question, all possible pieces of information are stored in an array.	This is an array of strings where the index of a string corresponds to the index of the answer in 'options' with which it is associated.

API/HTTP Communications

The frontend and the backend communicate through a set of predefined HTTP requests. The server listens on port 8000 and provides its functionality through the endpoints described below. These endpoints are called by the frontend through HTTP requests.

Endpoint for Inviting users

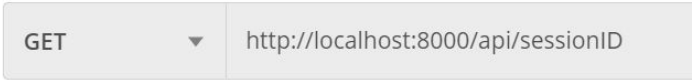
Endpoint: /api/inviteUsers
Purpose: Invite a list of users via SMS to take part in the survey.
Type: POST
Example Request:



Notes: The cohort value is used to determine which invite text should be sent to a given user based on whether they are a family member or a participant. Please use '1' for participants and '0' for family members.
The endpoint uses the GovNotify API to send the SMS messages, this process is explained in detail in the SMS Messaging with GovNotify section.

Endpoint to generate and send a session ID

Endpoint: /api/sessionID
Purpose: Generate a unique sessionID in order to group a user's responses anonymously.
Type: GET
Example Request:

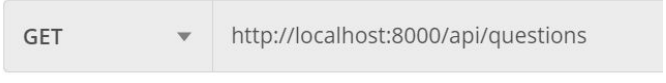


Example Response:

Notes: sessionIDs are generated by hashing the concatenation of the current time and the current increment of the sessionIndex variable with the SHA256 algorithm. This process is used to minimise potential collisions, meaning session IDs are unique for each session.

Endpoint to send the questions

Endpoint: /api/questions
Purpose: Get the list of questions.
Type: GET
Example Request:



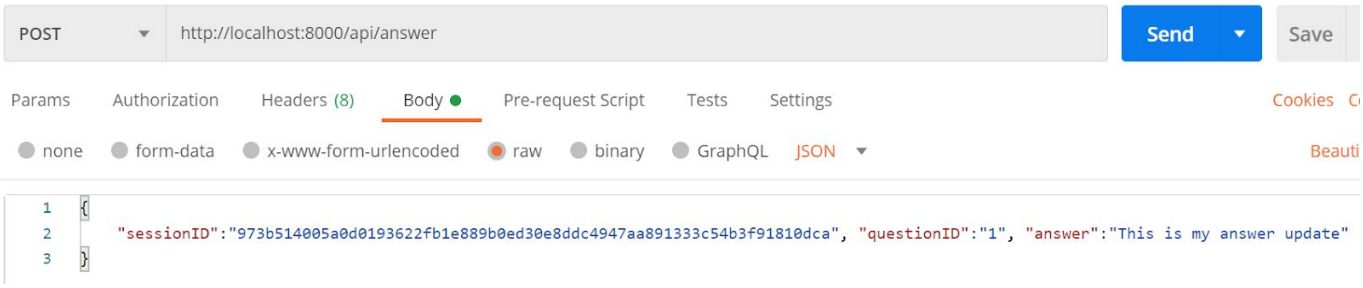
Example Response:

```
{
    "statusCode": 200,
    "questions": {
        "initialFamilyQ": 0,
        "initialParticipantQ": 2,
        "Qs": [
            {
                "id": "q1Family",
                "text": "Do you have any relatives aged 50 to 70 who have been invited for breast screening?",
                "options": [
                    "Yes",
                    "No",
                    "Not Sure"
                ],
                "nextQuestionId": [
                    1,
                    -1,
                    -1
                ],
                "longestPath": 2,
                "info": [
                    "",
                    "All women between 50 and their 71st birthday are invited for NHS breast screening every 3 years. <br><br>Find more information about <a href=\"https://www.gov.uk/breast-screening-info\" target=\"_blank\">NHS breast screening from gov.uk</a>.<br><br>You can also encourage your relatives to check their breasts and be aware of the symptoms of breast cancer. Find out more about the <a href=\"https://www.nhs.uk/conditions/breast-cancer/symptoms/\" target=\"_blank\">symptoms of breast cancer from the NHS website</a>.",
                    "All women between 50 and their 71st birthday are invited for NHS breast screening every 3 years. <br><br>If your relative has not been invited but think they should have been they can contact their local breast screening unit. <a href=\"https://www.nhs.uk/service-search/other-services/Breast-screening-services/LocationSearch/325\" target=\"_blank\">Find local NHS breast screening units</a>. <br><br>Find more information about <a href=\"https://www.gov.uk/breast-screening-info\" target=\"_blank\">NHS breast screening from gov.uk</a>. <br><br>Download the <a href=\"https://www.bhbss.nhs.uk/files/Breast_screening_The_Facts_In_bengali.pdf\" target=\"_blank\">NHS breast screening leaflet in Bengali (PDF)</a>."
                ]
            },
...
            ]
        }
    }
```


Notes: For more detail on this structure please read the section on Typescript interfaces.

Endpoint for storing a user's answer

Endpoint: /api/answer
Purpose: Store a user's answer to a question.
Type: POST
Example Request:

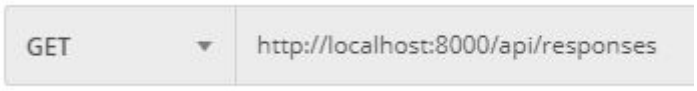


Notes: The answer provided by the user will be stored under the sessionID in the database. This technique has been used to provide anonymous data storage.

Endpoint for retrieving responses from the database

Endpoint: /api/responses
Purpose: Retrieve response data from the database in a formatted JSON object
Type: GET

Example Request:



Example Response:


```
{
    "statusCode": 200,
    "data": [
        {
            "Do you have any relatives aged 50 to 70 who have been invited for breast screening?": [
                {
                    "Yes": 12
                },
                {
                    "No": 5
                },
                {
                    "Not Sure": 3
                }
            ]
        },
        {
            "Did your relative go for screening when they were invited?": [
                {
                    "Yes": 6
                },
                {
                    "No": 3
                },
                {
                    "Not Sure": 3
                }
            ]
        },
        {
            "Have you ever been invited for breast screening?": [
                {
                    "Yes": 14
                },
                {
```


Notes: The returned JSON is structured in a readable format. All questions that are part of the survey (and the response data to those questions) will automatically be included in this JSON object.

Database Structure and Usage

We use a MongoDB database to store user response data. MongoDB was chosen for its scalability and use of noSQL data storage.

When a valid POST request is sent from the front end to '/api/answers' the server processes the data stored in the body of the request and either creates a new entry in the database to store the user's answer or updates an existing entry.

To determine whether an entry is created or updated the sessionID is checked, if the current session is a new session (identified by a sessionID that doesn't correspond to any entries in the database) then a new entry is created in the database. If the current session has a sessionID that matches an entry in the database then the responses within that entry are updated. This allows for answers to be dynamically stored and updated at any point in the survey so long as the users session remains active, meaning users can freely go back through the survey and change their answers if they wish.

Each session's response data is stored in the database as an object with this format:

```
{
    "sessionID":"5f222824d658f27c87a8854ae37bb432c190412eb869937b6a96dff99bed7286",
    "responses":[
        {
            "questionID":"q1Participant",
            "content":"1"
        },
        {
            "questionID":"q2Participant",
            "content":"2"
        }
    ]
}
```


“sessionID” stores the unique session ID string that is generated for that session by calling the ‘/api/sessionID’ endpoint.

Within the “responses” object are the individual responses to questions that a user has provided in that session. The “questionID” is the unique identifier (stored as a string) for each question as defined in the questions.json file on the server. The “content” is an integer value (stored as a string) that represents which response was given to the question as defined in the questions.json file.

SMS Messaging with GovNotify

Our system uses the GovNotify API to send SMS messages to users, inviting them to complete the survey. GovNotify was chosen as it is the standard system for government and government-associated services.

The management of the .env file is crucial to the functionality of the ‘notify.js’ script. This file contains two categories of information: 1. API key , 2. Template Keys. Please refer to the section on Setting up GovNotify to see how to configure the .env file.

API Key

The API key used to send messages must be a live key as this is the only key that can send messages to each user. This key will no longer exist once you navigate off the page.

Template Keys

Changing the content of message templates or the name of the message template:

The GovNotify dashboard provides the functionality to alter the message content. By navigating to ‘Templates’ in the dashboard and clicking ‘Edit’ the content of the message or the name of the template can be altered. If any new personalisation keys are added, changes need to be made to notify.js. Due to the length of the messages and the similarity between messages for the two cohorts, the same personalisation object is used for every text.

To add a new personalisation key enter a key-value pair into the personalisation object. The personalisation object must contain all required keys for each message.

personalisation = {

….

<new_personalisation_field>: <value_of_field>

….

}

Adding new message templates:

The code for sending messages was built with the mindset that we aim to keep text conversations to a minimum hence adding many different types of messages would require a new design. The current system does not support adding different types of messages, supporting two types of messages: an introductory message and a reminder.

The notify.js file exports a single function:

- sendMessage(user, type): sends a type of message either ‘intro’ or ‘reminder’ to a specific phone number read from the user object.

This function is not exposed to the user however is invoked when the ‘inviteUsers’ endpoint is triggered.

The user object contains information such as the phone number, name and cohort (participant or family) of the user. A message will be printed to the console if all messages were successfully delivered to each phone number.

Troubleshooting the messaging system:

1. Ensure that all relevant packages are installed using the node package manager.
2. Ensure input data is correctly formatted which can easily be done using regular expressions for a large data set i.e phone numbers contain 11 characters.
3. Ensure that the server is running.

In the future, the GovNotify API could be swapped out with another API service which allows for the sending of MMS messages, this could be used to send informative videos/infographics however the tradeoff would be using a non-government standard API. Another potential change that could be made in the future is using different types of personalisation depending on the message being sent. The current system is set up so that the same information is used to send all messages as the message content does not vary very much in terms of personalised content, however, as more message templates are created messages may need to be personalised differently.

Google Analytics

Google Analytics is integrated into the frontend to allow admins to view statistics about the usage of the web app. Google provides extensive documentation for Analytics so this documentation covers the specifics of our implementation of Analytics into this web app.

Data collected

This web app makes use of 2 major features of Analytics - content tracking and event tracking.

Content tracking provides insight into the usage of specific pages, allowing admins to track, for example, average time spent on each page and the percentages of people exiting the web app on a particular page.

Event tracking allows us to define custom events that Google Analytics then collects data on. The event categories and their sub-events that have been created are:

- "***Answer Changed***"
    - "*User Has Selected a Response That Is Different From Their Original Selection.*" - This event is triggered when a user selects an option for a response, but changes their response before submitting it, and is designed to allow some measure of users' indecisiveness for each question.
- "***Information Interaction***"
    - "*User clicked within the information section, possibly following a link to further information.*" - This event is triggered by a user clicking anywhere within the HTML divider that contains the information text on the question-specific information screens. This includes clicking on links that are contained within the text.
- "***Back Button Click***"
    - "*User clicked 'back' from* [initial page] *to* [previous page]." - This event is triggered by a user clicking the back button to navigate to the previous page they had been on. It enables admins to view patterns in which pages users may be most uncertain, or make the most mistakes. Please note that the initial page and the previous page are automatically inserted.

Sending data to Google Analytics

The React frontend uses an npm module called React-GA to simplify the process of collecting Analytics information.

- The function ReactGA.initialise(), shown below, is called whenever a user loads the web app, and is responsible for initializing a connection with Google Analytics and alerting it to a new user session. '`AnalyticsTrackingID`' should be replaced with the Tracking ID provided by Google Analytics, for more detail on setting up Google Analytics please refer to this section.

```
ReactGA.initialize('AnalyticsTrackingID');
```

- When the user navigates to a new page within the app, the function "sendGAPageUpdate" is called, which uses the two methods shown below to alert Google Analytics to a new page navigation event.

```
ReactGA.set({ page: fullPath });
ReactGA.pageview(fullPath);
```

- When a custom event is triggered, the relevant function of the form shown below is called.

```
ReactGA.event({
      category: "Back Button Click",
      action: "User clicked 'back' from " + originalQ + " to " +
newScreen + ".",
    });
```

Frontend Structure

Single page app

Our web app can be divided into three main sections: a Welcome Screen displayed to all users on arrival, a survey section which is used to display questions and information, and an end section with a screen thanking users for the time and confirming that their responses have been submitted.

The web app consists of a single page which is updated to display different content for each of these sections. This structure allows us to easily maintain a coherent layout and design for the pages and helps to improve the user experience. The updating of the display is done through the use of various components (see below).

There are 6 components that represent the main body of a screen and several further components used to create individual features of the pages. The 6 components that represent a screen are WelcomeScreenFamily, WelcomeScreenParticipant, ConnectionErrorScreen, QuestionScreen, InformationScreen, and FinishedScreen.

Components

Some components require "props" - properties, such as functions and variables, that are passed to the component from its parent component. As TypeScript has been used, each component that requires props has an "interface" that describes the names and datatypes of the properties it requires.

Welcome Screen Participant

The welcomeProps interface for this component stores a function called selectUserRole which is called when the user clicks continue, confirming that the user role identified for them is correct. The component also requires the MainButton component which is imported at the top.

This is one of the screen components and it contains the title of the app and a brief introduction to the project for participants.

Welcome Screen Family

The welcomeProps interface for this component stores a function called selectUserRole which is called when the user clicks continue, confirming that the user role identified for them is correct. The component also requires the MainButton component which is imported at the top.

This is one of the screen components and it contains the title of the app and a brief introduction to the project for the family members of participants.

Question Screen

This takes 5 pieces of data through the props interface:

- nextScreen() is the function for moving to the next screen.
- previousQuestion() is the function for moving to the previous screen.
- questionText() is the text which makes up the body of the question.
- questionResponses is an array of strings containing the possible responses to the question.
- progressPercentage is the current progress of the user through the journey for the progress bar.

The question screen requires several other components in order to function correctly. These are imported at the top of QuestionScreen.tsx and are listed here: MainButton, ProgressBar, GoBackButton, and QuestionRadio. The component also imports ReactGA for Google Analytics integration.

There is a function called handleClick that allows users to select a response to a question and updates Google Analytics if they've changed their response.

The question screen displays the question text and the available options are presented as a radio selection, allowing the user to pick one response.

The page has a Continue button which runs the function to move to the next question when clicked and is disabled while no response has been selected.

The page also has a ‘Go back’ button which runs the function to move back to the previous answered question when clicked and a progress bar which indicates to the user how far they are through the survey.

Information Screen

The information screen takes 4 pieces of data as props:

- nextScreen() is the function for moving to the next question.
- previousQuestion() is the function for moving to the previous question.
- informationText is the text to be displayed. This is a string of HTML code that is parsed by the component and then displayed to the user.
- progressPercentage is the current progress of the user through the journey for the progress bar.

The information screen requires several other components in order to function correctly. These are imported at the top of InformationScreen.tsx and are listed here: MainButton, ProgressBar, and GoBackButton. The component also imports ReactGA for Google Analytics integration and ReactHtmlParse for displaying text with HTML formatting.

The purpose of this screen is to display the information text, with HTML formatting, to the user.

The page has a Continue button which runs the function to move to the next question when clicked.

The page also has a ‘Go back’ button which runs the function to move back to the previous answered question when clicked and a progress bar which indicates to the user how far they are through the survey.

Finished Screen

The finished screen takes no props and displays a simple message thanking users for their time and telling them their answers have been saved. It also includes the progress bar showing 100% completion, the ProgressBar component is imported at the top of FinishedScreen.tsx.

Connection Error Screen

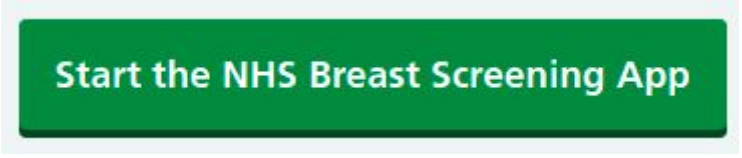
The connection error screen takes no props and displays a simple message to the user, informing them that the app has been unable to establish a proper connection to the server.

MainButton

The main button takes 3 props: a string called text which contains the text to display in the button, an optional boolean value called disabled which is used to disable the button until a question has been answered and a function called onClick() (return type void).

This button uses the NHS service manual component and styling to display a primary button with the text passed as a prop.

I include an example of the button where the text passed is ‘Start the NHS Breast Screening App’.



GoBackButton

This component takes a function called onClick() (return type void) through its Props interface and uses the NHS service manual component and styling to display a button saying ‘Go Back’.

Question Radio

This component takes 4 pieces of information through its Props interface. These are a string called text, a string called value, an optional boolean variable called checked and an optional function called onChange. This component is used to create a radio for an answer to a question. It uses the NHS service manual components for styling.

Header

The header component implements the NHS transactional header from the Service Manual.

Footer

The Footer component contains several links, in accordance with the footer from the NHS Service Manual.

Progress bar

The progress bar takes one piece of information through the ProgBarProps interface, a number representing the percentage of the user’s progress through the survey. A progress bar is then generated with 3 styled divs.

NHS design library

To meet the client's specification our project is fully integrated with the NHS frontend kit, this is accessed using the node module nhs-frontend, to read more about the NHS frontend style guidelines please refer to their website.

The NHS components are included in the client/nhsukscss.scss file which is compiled using gulp (configured in client/gulpfile.js) to create nhsukscss.css.

This allows components created with the classes specified in the NHS Service Manual to have the correct styling and behaviour.

State variables

In order to track a user’s progression through the survey, the frontend manages several state variables for the App.tsx component, which are similar to global variables for the web app. These store the following information and have the following default values.

Variable	Purpose	Default
questions	This stores the JSON questions object so that the questions can be accessed by other functions in the program.	The default value is a dummy question used to indicate that the questions have not yet been obtained from the server. This is overwritten with the correct set of questions once the page has loaded.
currentQuestion	This stores the JSON question	The default value is the first

	item of the question that the user is interacting with currently. To understand the structure of the question items please refer to Question Architecture Design.	question on the family path.
answeredQuestions	This stores a list of JSON question items that have already been answered by the user. This information is used to facilitate the 'Go Back' button functionality and to calculate the user's progress for the progress bar.	The default value is an empty array of question items, denoting that no questions have been answered.
questionOrInfo	This is a global boolean variable which is used to decide whether to display a question or the associated information. The information is stored alongside the question item (please refer to Question Architecture Design for more detail).	The default value is 'true', denoting that a question should be displayed rather than its associated information.
startMiddleEnd	This variable is used to manage the different screens displayed by the web app. As outlined in Single Page App structure, the web page is split into three sections, the Welcome Screen, the Questions Screen and the Finished Screen.	The default value is 'start' to ensure that users are greeted with the Welcome Screen when they first enter the page.
sessionID	This stores a unique string for a user's session. This string is sent to the server along with the user's answers to the questions to facilitate anonymous data storage and processing.	The default value of this variable is undefined, this is updated when the frontend receives a sessionID from the server.

Typescript interfaces in App.tsx

We elected to use a strongly typed version of JavaScript to develop this system in order to ensure predictability when data is passed between functions, and to avoid type coercion issues. Our project includes several 'custom data structures' which we have defined as interfaces in order to be used in the program.
There are also some "Prop" interfaces specific to components, these are described in the Components section for clarity.

Name	Purpose	Structure
SessionIDResponse	The expected format of a valid JSON response from the server to a request for a sessionID.	`sessionID: string;`
QuestionResponse	The expected format of a valid	`questions: QuestionsObject;`

	JSON response from the server to a request for the QuestionsObject. 'questions' is expected to be of the type 'QuestionsObject' defined below.	

QuestionItem	Used to store the details of a single question. Each question has an id; some text that stores the question; an array of possible options to answer with; a corresponding array of ids for the next question depending on the response; information about the longest path from here (used in the progress bar); an array of strings of information that can be displayed to the user once the question is answered which also corresponds with the response options; and an optional field to store the response that the user selected (necessary in order to allow users to revisit questions).	`id: string;`
		`text: string;`
		`options: Array<string>;`
		`nextQuestionId: Array<number>;`
		`longestPath: number;`
		`info: Array<string>;`
		`userResponse?: string;`

QuestionsObject	This stores an array of question items along with the indices of the initial question for each of the user journeys. The indices (initialFamilyQ and initialParticipantQ) are used for accessing the right initial question from the array depending on the cohort to which the user belongs.	`initialFamilyQ: number;`
		`initialParticipantQ: number;`
		`Qs: Array<QuestionItem>;`

Functions

App.tsx has a functional component called App that manages the application. Within this, several other functions are defined to perform various utilities. These are all explained below, in the order that they appear within the code.

49 - 62	Create a default QuestionsObject to assign to the currentQuestion state variable. It is used in screenToDisplay() to check that getQuestionsObject() successfully retrieved the questions from the server.
65 - 80	Declare and initialise the State Variables.
84 - 106	Define the sendAnswer function that sends a user's answers to the server.
109 - 124	Define the getSessionId function which requests a unique session ID from the server.
127 - 140	Define getQuestionsObject(), a function which requests the questions object from the server.
143 - 160	Define sendGAPageUpdate, a function for alerting Google Analytics to a change of the current page and useEffect(), which is called when the App component is loaded and is responsible for initializing the connection to Google Analytics and loading data from the server.
163 - 186	Define continueButtonPressed, a function which determines and enacts the actions necessary when the continue button is pressed, based upon the current screen. If the current screen is a question screen, it sends the user's answer to the server using sendAnswer() and fetches the information to display to the user if their answer has associated information. It also updates Google Analytics with the current page (using sendGAPageUpdate). For a continue button on an information screen, these steps are not required and so are not enacted. In either case, if there is no further information to be shown it calls the nextScreen function.
189 - 202	Define the nextScreen function, which updates the array of answered questions. If there are more questions in the survey it updates the current question (stored in a state variable) and updates Google Analytics (using sendGAPageUpdate). If the end of the survey has been reached, the function updates the startMiddleEnd state variable to indicate that the FinishedScreen component should now be displayed and updates Google Analytics (using sendGAPageUpdate).
205-230	Defines the PreviousQuestion function which checks whether the current screen is a question or information. If it's information then it sets the newScreen to be the question screen that led to the information displayed. If the current screen is the first question the newScreen is set to show the welcome screen again. If the current screen is another question in the survey then the function sets newScreen to be the previous question. In every case, the function sends an appropriate update to Google Analytics using sendGAPageUpdate.
233 - 242	Defines the function selectUserRole, which sets the state variable currentQuestion to be the initial question of the correct journey. It also updates Google Analytics (using sendGAPageUpdate) and sets the startMiddleEnd state variable to 'middle', indicating that the user has moved from the welcome screen to the questions.

245 - 308	This section defines the screenToDisplay function. This returns the code needed to construct either a welcome screen, a question screen, an information screen or a finished screen, in the form of a react component.
311 - 323	The return value of the App() function is a JSX component containing the HTML to display the interface of the web app. These lines combine the container, including header, main content divider and footer, for all pages with the react component returned by screenToDisplay().