React Web App – Decisioning Dashboard

Benjamin Roberts

Level 4 Software Development

Contents

[Table of Figures 4](#_Toc174119138)

[Introduction & Requirements 6](#_Toc174119139)

[Project Description 6](#_Toc174119140)

[Acceptance Criteria 6](#_Toc174119141)

[Design 7](#_Toc174119142)

[Existing product for inspiration 7](#_Toc174119143)

[Wireframes 8](#_Toc174119144)

[Initial Draft for page template 8](#_Toc174119145)

[Figma Wireframe 9](#_Toc174119146)

[Pages to develop 10](#_Toc174119147)

[Development 11](#_Toc174119148)

[Backend Setup – for supporting functionality 11](#_Toc174119149)

[Why one was required 11](#_Toc174119150)

[Initialising Node.js Express 12](#_Toc174119151)

[Testing connectivity 13](#_Toc174119152)

[Front End – React 14](#_Toc174119153)

[Top Navigation Bar 14](#_Toc174119154)

[Side Bar Navigation for longer pages 17](#_Toc174119155)

[SQL - Search for Recent Applications 20](#_Toc174119156)

[SQL misuse and Personal Data (PII) Protection 24](#_Toc174119157)

[Data – Decoding for Business Users 25](#_Toc174119158)

[Environments configuration 35](#_Toc174119159)

[Final Tech Stack 37](#_Toc174119160)

[Feedback - Review with Employer 38](#_Toc174119161)

[Final Application Pages 39](#_Toc174119162)

[Home.jsx 39](#_Toc174119163)

[SummaryData.jsx 39](#_Toc174119164)

[PremiumValue.jsx 40](#_Toc174119165)

[RawData.jsx 40](#_Toc174119166)

[Conclusions 41](#_Toc174119167)

[Project Outcome 41](#_Toc174119168)

[Future Additions 41](#_Toc174119169)

[References 42](#_Toc174119170)

[Appendix 44](#_Toc174119171)

[Initial Backend attempt, with React & Node.JS using concurrently 44](#_Toc174119172)

[Hosting in Azure Static Web App 45](#_Toc174119173)

[Confirming access 48](#_Toc174119174)

# Table of Figures

[Figure 1- Requesting screenshots of a similar existing product from a colleague 8](#_Toc174119082)

[Figure 2- Initial draft Wireframe drawn on Whiteboard 9](#_Toc174119083)

[Figure 3 - Figma Wireframe of a Home Page for the React app 10](#_Toc174119084)

[Figure 4- Importing "mssql" into a React Component called "Home.jsx" 12](#_Toc174119085)

[Figure 5- Compile error from importing the "mssql" library 12](#_Toc174119086)

[Figure 6 - Initialising an Express app based on a templete 13](#_Toc174119087)

[Figure 7 - Install http-errors dependency 13](#_Toc174119088)

[Figure 8- "user.js", a default route provided by Express 14](#_Toc174119089)

[Figure 9- Postman HTTP Request showing the expected response from the "users.js" route 14](#_Toc174119090)

[Figure 10-Top Bar Navigation for all pages on the application 15](#_Toc174119091)

[Figure 11- HTML and React Code for the Navigation Bar 15](#_Toc174119092)

[Figure 12 - NavBar Component placed into the main App.JSX Component 16](#_Toc174119093)

[Figure 13 - CSS code used to make the Top Navigation bar scroll with the user 16](#_Toc174119094)

[Figure 14 - Top Bar Navigation scrolls with user 17](#_Toc174119095)

[Figure 15 – Webpage with enough content to require the user to scroll down to view different sections 18](#_Toc174119096)

[Figure 16- Side Bar HTML code, with <a> to provide the links 18](#_Toc174119097)

[Figure 17- <h2> tags with Id's allowing the side navigation bar to link to them 19](#_Toc174119098)

[Figure 18- Original CSS, with an issue on the "top" property not working as expected 19](#_Toc174119099)

[Figure 19- Altered CSS, with unnecessary properties removed & issue with "top" property fixed 20](#_Toc174119100)

[Figure 20 – Final Side Bar for the PremiumValueData page, with links to each section and sub-section 20](#_Toc174119101)

[Figure 21- Required imports for this new "db.js" file 21](#_Toc174119102)

[Figure 22- Configuration object 21](#_Toc174119103)

[Figure 23 - SQL Query in use 21](#_Toc174119104)

[Figure 24 - Code behind the "db.js" file that makes the Database Connection possible 22](#_Toc174119105)

[Figure 25- Calling the Back End with fetch, wrapped in the "UseEffect" React function 23](#_Toc174119106)

[Figure 26 - CORS Error message 23](#_Toc174119107)

[Figure 27- Code added to "db.js" to resolve CORS Policy errors 24](#_Toc174119108)

[Figure 28 - Rendered Database results 24](#_Toc174119109)

[Figure 29- Permissions applied to the SQL Integration account 25](#_Toc174119110)

[Figure 30- Definition of E1B08 from Experian (2024) documentation 26](#_Toc174119111)

[Figure 31 - Definition of AccountStatus from Experian (2024) documentation 26](#_Toc174119112)

[Figure 32- Installing csv-parser 27](#_Toc174119113)

[Figure 33- Error attempting to load a dependency directly into React for my lookup function 27](#_Toc174119114)

[Figure 34- Router.Get method for the backend to accept a search parameter 28](#_Toc174119115)

[Figure 35 - Endpoint logic showing the lookup function being used & sending this back to the calling application 29](#_Toc174119116)

[Figure 36 - Code for Lookup functionality 30](#_Toc174119117)

[Figure 37- OnClick property using the lookup function 31](#_Toc174119118)

[Figure 38 - Lookup prompt presented to the user for E1B08 as seen in Firefox 32](#_Toc174119119)

[Figure 39- Lookup prompt presented to the user for E1B08 as seen in Microsoft Edge 33](#_Toc174119120)

[Figure 40- Switch case to translate AccountStatus into a business friendly value 34](#_Toc174119121)

[Figure 41 - CaisRow component return statement, showing the translated value in use 34](#_Toc174119122)

[Figure 42- React JSX code showing the rendering of the CaisRow components via the .map() method 35](#_Toc174119123)

[Figure 43 - Rendered React application showing the business friendly language used instead of the original values 35](#_Toc174119124)

[Figure 44 - Hardcoded URL within the React Application 36](#_Toc174119125)

[Figure 45 - .env file added to allow dynamic configurations without altering the raw code 36](#_Toc174119126)

[Figure 46 - Altered code to use the newly created .env file 37](#_Toc174119127)

[Figure 47 - Diagram showing the Tech Stack of the full application 38](#_Toc174119128)

[Figure 48 - Teams Call arranged to discuss Apprenticeship progress 39](#_Toc174119129)

[Figure 49- Home Page (final) 40](#_Toc174119130)

[Figure 50 - Summary Data page (final) 40](#_Toc174119131)

[Figure 51- Premium Value page (final) 41](#_Toc174119132)

[Figure 52- Raw Data page (final) 41](#_Toc174119133)

[Figure 53- Initial Backend npm start logic 45](#_Toc174119134)

[Figure 54 - Setting up an Azure Static Web App to host the React application 46](#_Toc174119135)

[Figure 55 - The React App home page as deployed to the Azure Static Web App 48](#_Toc174119136)

[Figure 56 - Confirmation a user can access the deployed React App 49](#_Toc174119137)

# Introduction & Requirements

With my team being responsible for developing and maintaining our Credit Decisioning system, we require various kinds of Management Information (MI) and views of the data used by the platform (both in the aggregate and within individual applications). This is also important for monitoring the performance of the platform.

Today, we use various tools to monitor and report on the performance of the system e.g.:

* Ad-Hoc analysis done via querying a SQL Database
* Summarised Microsoft Excel Reports
* Microsoft Power BI reports
* Manually extracting and reviewing raw data (typically via SQL)

All these tools are useful for their specific purposes but what they don’t do is give us a presentable view of the low-level data used in making a decision on an application. We could license other software for this purpose e.g. the Credit Bureau providers Experian, TransUnion and Equifax all provide software to present the data they hold on an applicant. These however require software licenses to be purchased. I believe we could use the React Framework to build a similar tool that would be more bespoke to our business needs.

## Project Description

Using the React Javascript Framework, design and develop a simple Web Application to:

1. Leverage the Java Decisioning Application I developed in a previous project & it’s logged output (Link: [GitHub - Java Credit Decisioning Program](https://github.com/btr6566/qam1_java_decisioning_app))
2. Display a list of recent applications made to the Credit Decisioning system
3. Allow a user to click on a specific application to view the data used for it in more detail
4. Provide reference material for the data definitions of the data used (i.e. a Data Dictionary function)

## Acceptance Criteria

1. Wireframe of the design of the website is provided
2. Final product aligns to the wireframe
3. Intuitive user interface created & evidenced via feedback from technical & non-technical stakeholders
4. The web app is able to query a database where logs from the Decisioning program/software are retained
5. Low level detail of the data used in the application, including:
   1. Data used for running Scorecards + their values
   2. Summary data on an applicants Credit file
   3. Raw data used to create the above summary data

# Design

To give myself a starting point, I referred to the Software Development Life Cycle, which as “design” as a key starting point for build a piece of software (Amazon AWS, 2024). With the requirements already defined, I needed a design to work towards.

## Existing product for inspiration

From my time in the workplace, I was aware of a similar product that provided similar functionality to what I was aiming to achieve in this project. I believed it would be worthwhile to examine this product as a source of inspiration for how to approach my project.

I did not have access to this product myself, but I knew a colleague who did. I sent them a message over the Microsoft Teams application asking if they could share some screenshots of what they see within this product. They provided me with 5 screenshots.

A screenshot of a computer

Description automatically generated

Figure 1- Requesting screenshots of a similar existing product from a colleague

I’ve not included the provided screenshots in this report & have removed the majority of the previews from the figure above. This information can be considered trade secrets, so this was done to protect the privacy of the stakeholders involved.

## Wireframes

With some inspiration from an existing system Vanquis Banking Group already use, I started with a draft wireframe and then expanded upon it with a 2nd iteration.

### Initial Draft for page template

To keep a simple process to start, I drew a basic template on a white board for that I had in mind for a “home” page:

A white board with a drawing on it

Description automatically generated

Figure 2- Initial draft Wireframe drawn on Whiteboard

Doing this simple approach gave me a way to get a starting point for what I wanted the layout of my application to look like, without having to decide on specific details (like styling).

### Figma Wireframe

The whiteboard draft was useful to get a foundation, but I wanted to go into a little more detail before I started creating the React Application. This meant I needed a more detailed wireframe.

Figma is a solution I was familiar with, as our internal Marketing team have shared wireframes they’ve created for internal projects with me previously in the past. I believed this made it a good tool to look at for developing my wireframe into something ready for development.

Below is what I created for the Home page. The idea was to use this as a template for the design of the remaining pages.

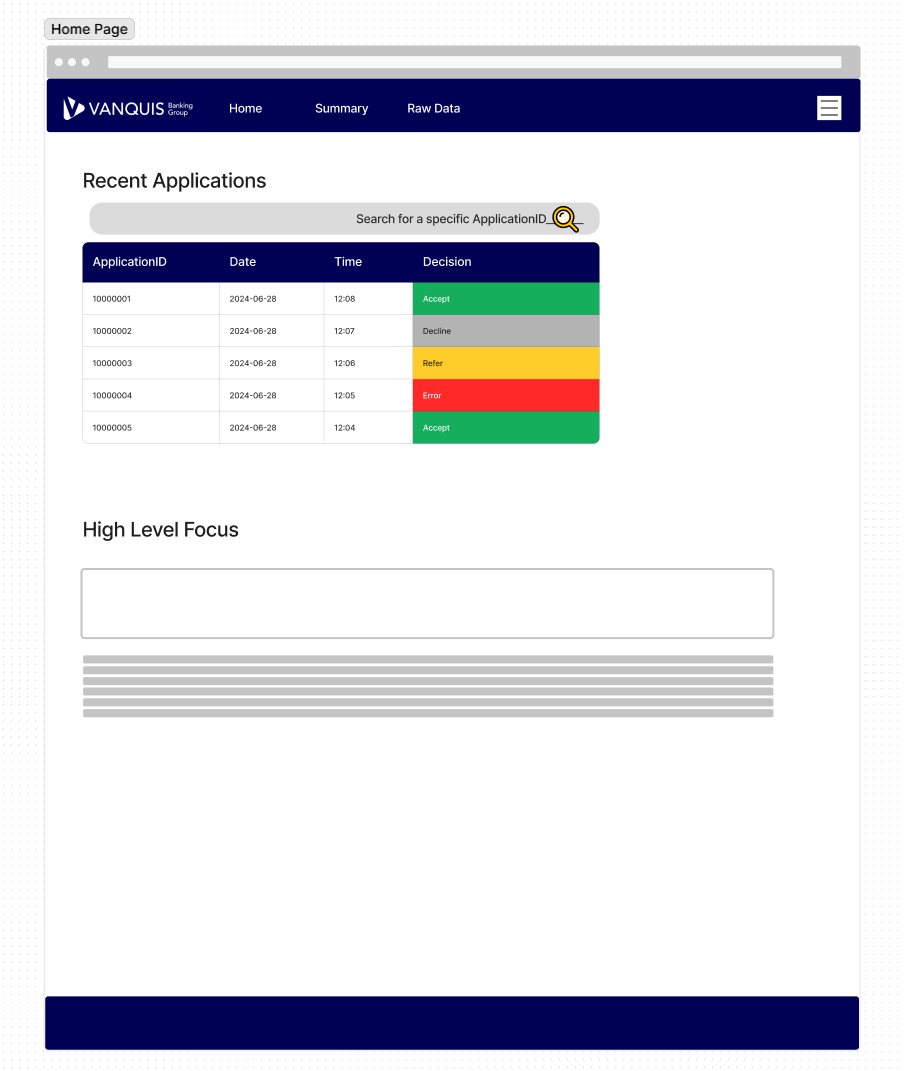


Figure 3 - Figma Wireframe of a Home Page for the React app

One key change I made from my initial draft wireframe was to use a header Navigation Bar instead of a left-hand side bar. This simplified the layout of the website and meant the functionality presented to the end user could be focused here, allowing them to navigate the core of the application more easily.

For styling, I consulted our Vanquis Banking Group branding guidelines. I wanted something that looked like it could really be in use in the organisation & to do this effectively, it was critical to follow the branding guidelines to provide a professional appearance to corporate material (Iconic Digital Marketing Agency, 2024).

### Pages to develop

With a wireframe template ready for the basic design I wanted, I was ready to think about what pages I wanted in the applications. Below are a list of pages to build, along with some information behind their meaning

* **Home** – A simple home page showing the most recent applications logged in the database
  + This will be the landing page a new user is taken to when they open the application
  + This will be where the most recent applications are displayed to the user
* **Summary** – A page where a user could view what we refer to as the “Summary” data provided by Experian (2024), which is an aggregated view of the data they hold on a person’s Credit File
  + Within a business context, the data points in this block of data are primarily used within the Credit Decisioning processes that decide if an applicant should be given a line of credit
* **Premium Value** – Similar to Summary, but provides significantly more data points
  + This page’s use case is like the Summary page as a result
  + In the business view, this is called “Premium Value Data” as they are additional blocks of data Experian provide for an extra fee.
  + They are very rich data source for use in Credit Decision, as they can be used to identify problematic or positive trends on a person’s credit file, often to a better extent then the Summary data alone
  + There are also product specific data blocks, which allow a lender to check for behaviour on specific products where trends would be more meaningful e.g. status of Mortgage and Utility accounts on a persons credit record
* **Raw Data** – This should contain the unaggregated data used to derive all the datapoints displayed in the Summary and Premium Value pages

# Development

## Backend Setup – for supporting functionality

Some of the requirements I set required a full Backend to be created to implement the functionality required. This included:

* Ability to connect to a database to retrieve application data (an Azure SQL Server Database specifically was used in the project)
* Searching an External file for definitions of data variables

### Why one was required

Initially I tried importing the “**mssql**” package directly into the React component to meet the 1st use case of connecting to a database to retrieve application data. When doing that however, I faced 26 compiler errors in React. These came about purely from importing the library for use:

A screenshot of a computer program

Description automatically generated

Figure 4- Importing "mssql" into a React Component called "Home.jsx"

A screenshot of a computer program

Description automatically generated

Figure 5- Compile error from importing the "mssql" library

One thing I didn’t understand was what this “**Polyfill**” concept the error messages referred to. That felt like the 1st step to resolving the error, so I did some quick research. I found that **Polyfills** are pieces of code to provide modern functionality to older browsers, at the expense of functionality and performance (Mozilla, 2024a). From the context of the error message, I can see that this means that the “crypto” module mentioned in the error message is no longer automatically included, so creating a **polyfill** is necessary to resolve.

Attempting to create one of these polyfills did not resolve my issue. From further research I concluded that it would be significantly easier to set up a back-end application that the main React application could call over HTTP e.g. via the fetch method. This allows offloading complex logic into a separate app and avoids the need for complicated polyfill logic.

### Initialising Node.js Express

After some trial and error, I came to think that it would be easiest (to develop and maintain) a separate Node.js Backend and use HTTP calls within the main React application to get the necessary data. That creates a clear distinction between front-end and back-end, plus means they can be maintained separately. I found the Express framework from some research, which provided the framework I needed and could be generated with a command line function, shown in the figure below (Express, (n.d.).

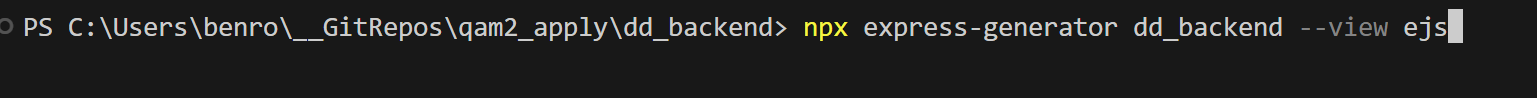


Figure 6 - Initialising an Express app based on a templete

This did require an additional dependency via a library called “http-errors”, as shown in the figure below. To resolve this, I ran another npm install command to add this into my repository.

A computer screen shot of a program

Description automatically generated

Figure 7 - Install http-errors dependency

Once this package was installed, I was able to start initialise the Express application.

## Testing connectivity

Once the Express application was initialised, the next step was to test connectivity. By default, the application starts with a basic “users.js” route that can be used for this (the routes being what Express uses to stand up resources with different functionality).

A screenshot of a computer program

Description automatically generated

Figure 8- "user.js", a default route provided by Express

When a HTTP call is made to this “users” route, it will simply send a text string with the value “respond with a resource”, back to the requesting service. This simplicity makes it useful to test that the application can be communicated with.

After starting the Express application with “npm start”, I made use of a software called Postman to test connectivity. This is a tool made for API Development and is primarily used for sending requests to websites to get a response. Automated tests could then be run as test cases on the response received. I use this software almost daily in my Day to Day role, so I was certain it was an effective choice for this test.

Show below is a Postman HTTP request to conduct the test. We can see the locally hosted express app did indeed respond with the expected value.

A screenshot of a computer

Description automatically generated

Figure 9- Postman HTTP Request showing the expected response from the "users.js" route

## Front End – React

Details of specific functionality developed for the Front End are given below:

### Top Navigation Bar

Critical to the application was a user navigation interface. I wanted something simple that was always accessible by the user no matter where they were on the page, hence a navigation bar at the top of the screen. Using my wireframe as a reference, I created the below:

A screenshot of a computer

Description automatically generated

Figure 10-Top Bar Navigation for all pages on the application

The below figure shows the HTML and React code that creates this navigation bar, called “NavBar.jsx” within the files. This is designed to return a simple <nav> html object to the page where this component is rendered, allowing it to be reused across the application. Link objects from the “react-router-dom” library are used to link to the different pages of the application, which allows the user to switch pages without triggering a refresh of the web browser used (React Router, n.d).

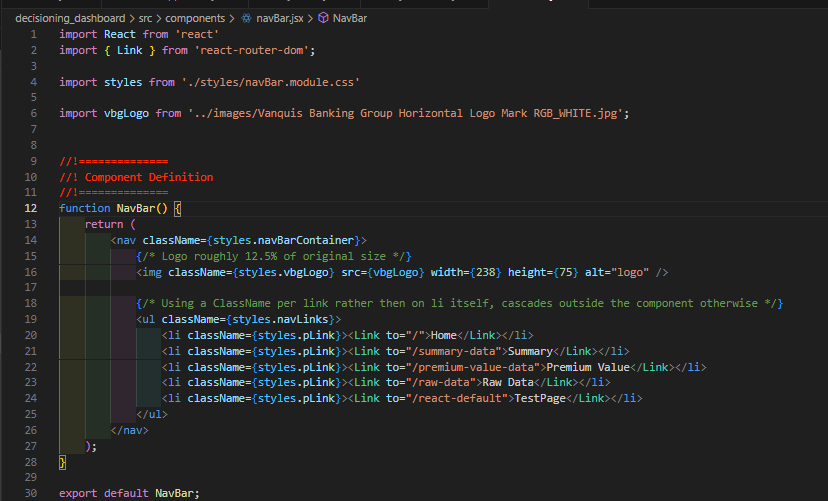


Figure 11- HTML and React Code for the Navigation Bar

This “NavBar.jsx” component is then placed within the root “App.jsx” component of the React Front End application, allowing it to be displayed across all pages. It is placed within the parent <BrowerRouter> component provided by “react-router-dom” library to provide seamless navigation between pages.

A screen shot of a computer

Description automatically generated

Figure 12 - NavBar Component placed into the main App.JSX Component

To meet the desire to have this navigation bar visible to the user no matter how far they scroll on the page, I added CSS logic. The <nav> html tags that contain the Navigation bar has the “sticky” position property applied to it, which enables this feature (W3Schools – CSS Position, 2024).

A screenshot of a computer program

Description automatically generated

Figure 13 - CSS code used to make the Top Navigation bar scroll with the user

Below is an example of this feature working as expected. In this example, a user has scrolled down one of the pages while viewing a table component. Despite this however, we can see the top navigation bar has stayed at the top of the users window, allowing them to access it immediately.

A screenshot of a computer

Description automatically generated

Figure 14 - Top Bar Navigation scrolls with user

### Side Bar Navigation for longer pages

One issue I felt while developing the pages that displayed the detailed application data to the end user was that displaying all this data resulted in very long web paged e.g. shown below is a page called “Premium Value Data”, which displays various data points provided by Experian (2024). This section of the data has multiple sections, each with many variables. The React application renders these as HTML tables, but the number of variables in each section results in a long web page, making it difficult to navigate.

A screenshot of a computer

Description automatically generated

Figure 15 – Webpage with enough content to require the user to scroll down to view different sections

To address this, I went back to the side bar navigation idea I had on my original wireframe. Shown below is a snippet of the HTML code that creates this side bar. As the links are bespoke to the current webpage, I chose to stick to simple HTML rather then creating specific React components. This works by creating a container <div> tag to store all the links the user can click, which allows formatting to be applied to that section of the web page. Links are then ordered by creating a bullet point list with the <li> tags. The <a> these contain are what actually provide the links.

A screen shot of a computer program

Description automatically generated

Figure 16- Side Bar HTML code, with <a> to provide the links

Each <a> tag above has an id given for it’s href property, which tells the tag to take the user to the object with the same name as what’s in the href property. Shown below are examples of the corresponding HTML tags those links connect to:

A computer screen shot of a program

Description automatically generated

Figure 17- <h2> tags with Id's allowing the side navigation bar to link to them

With the HTML in place, I added a new CSS module file to use for this navigation bar, which could be re-used for any page that needed one. My initial CSS code for this can be seen in the figure below. With this I was attempting to have a similar “sticky” function as the top bar navigation covered above i.e. no matter how far the user scrolled, the links would stay visible to the user. This did not work with the CSS in the figure below however.

A screenshot of a computer program

Description automatically generated

Figure 18- Original CSS, with an issue on the "top" property not working as expected

To address this issue, I looked for a reference on how to apply this feature. I found that my issue was the top property was missing a unit of measurement (Mozilla, 2024b). After adding “px” to the tags (indicating pixel measurements). The scroll feature worked as expected. I was also able to refactor the CSS code to remove properties that were redundant to the feature I wanted here.

A screenshot of a computer program

Description automatically generated

Figure 19- Altered CSS, with unnecessary properties removed & issue with "top" property fixed

These steps provided a way to provide section links on the page, allowing the user to navigate quickly to various sections. The final version, when applied to the Premium Value Data page appears as such:

A screenshot of a computer

Description automatically generated

Figure 20 – Final Side Bar for the PremiumValueData page, with links to each section and sub-section

### SQL - Search for Recent Applications

Meeting the requirement to show recent applications logged to the backend database required making a SQL connection. The Node.JS Express Back End created earlier was used for this. To get started, I needed an Endpoint I could call with the Front end to get the data. An Endpoint represents a point where 2 applications can talk to each other (Yasar, K, 2024), so creating one in my backend is required for it to be able to communicate with the front end.

I created a new JavaScript file called “db.js” under the “routes” folder the Express framework uses by default to store Endpoints. I used the existing “users.js” that is initialised by default as a reference to start. I then imported the “**mssql**” node module I needed to make the database connection.

A screenshot of a computer

Description automatically generated

Figure 21- Required imports for this new "db.js" file

To make this connection work, I next needed something to store the configuration details in. The reference of the mssql library in use created a “config” object (Microsoft, 2024), so I did the same. To start with, I left this as a hard-coded object so I could get this working on a localhost first:

A screenshot of a computer program

Description automatically generated

Figure 22- Configuration object

A SQL Query was also needed, which I defined as a string within the “db.js” code:

A screenshot of a computer program

Description automatically generated

Figure 23 - SQL Query in use

Finally, the bulk of the logic occurs within this “router.get” method. This will listed for all HTTP traffic that is intended for the Endpoint in question e.g. “db.js” in this case. The “sql.connect” method within is part of the imported “mssql” library and is what makes the database connection possible. This reads the “config” object defined earlier for connection settings.

The “request.query” method then actualy executes the passed in SQL statement. The 1st IF-THEN-ELSE Logic exists for catching a connection error to the database (i.e. something went wrong communicating to the database), while the 2nd logic handles any errors in the query itself e.g. reference column does not exist.

Assuming no errors are returned, then a object called “recordset” contains the data returned by the database. This is then sent back to the requesting service via the “res.send()” method. If there is an error, the “err” object will be populated and that will be sent instead.

A screen shot of a computer program

Description automatically generated

Figure 24 - Code behind the "db.js" file that makes the Database Connection possible

With the Back End logic built, the Front End can now call this via the “fetch” method within native JavaScript to make a HTTP call to the backend, which retrieves the data. This method is wrapped in the “useEffect” function from the React library, as not including caused an issue where the data would not be displayed when received. The “useEffect” function forces React to re-render the webpage when the “fetch” method completes.

A screen shot of a computer program

Description automatically generated

Figure 25- Calling the Back End with fetch, wrapped in the "UseEffect" React function

One problem I ran into while testing this is that the fetch method would not succeed. Looking at the Web Browser console, I saw errors indicating the HTTP request to the backend was being blocked by CORS policy:

A screenshot of a computer

Description automatically generated

Figure 26 - CORS Error message

I wasn’t sure how to handle this, so I did some research. I found that these CORS errors are a common problem. I found a suggested fix on Stack Overflow (2014) to add in some response headers to the response sent by the “db.js” endpoint in my backend, which did correspond to the wording within the error message I was getting. Based on this, I added the below lines into my “db.js” code:

A screen shot of a computer program

Description automatically generated

Figure 27- Code added to "db.js" to resolve CORS Policy errors

Once that change was made, the call to the Back End succeeded and the database results rendered in the React application as expected:

A screenshot of a computer

Description automatically generated

Figure 28 - Rendered Database results

### SQL misuse and Personal Data (PII) Protection

One consideration for the database was connection was the potential for PII to be exposed, as using the Back End Express application. The data used within the program needed to be used with caution, as the Experian dataset used requires the below to be able to search a person’s credit file:

* Full name of the person in question
* Date of Birth
* At least the current address of the person in question
* Previous addresses are also often needed if the person in question has resided at their current address for less than 3 years

(Experian, 2024)

Each of these Data Points are protected under the UK's Data Protection Act 2018, which also codifies the GDPR into UK Law (Data Protection Act, 2018). This becomes an issue with the Experian Data used as it retains this information in the response data their API sends when used. As the idea of this application was to display all application data, this creates a conflict. As such, the program only displays data that does not contain PII data. PII Data is not rendered at all.

Also, to avoid any misuse of the database that could put this information at risk, I applied minimal permissions to my SQL connection. Core to this was the user account I created in the database to enable access, called “node\_integration\_user”. I applied only SELECT permission on this user and only to SQL Tables and Views actually required by the program.

A screenshot of a computer

Description automatically generated

Figure 29- Permissions applied to the SQL Integration account

### Data – Decoding for Business Users

One issue I had to solve was to consider how end users would understand the data presented to them by the application. There was however a complication caused here by the data sources used for this application, as a key part of the data used comes from data provided by Experian’s API for accessing data they hold on a person (Experian**,** 2024). This challenge manifests itself in 2 ways:

**Scenario 1**: Names of variables are highly codified to make their meaning unclear at 1st view e.g. E1B08 is a variable returned by the Experian API. Without it’s documentation as shown below however, this gives no indication as to what the variable means in business terms.

A screenshot of a computer

Description automatically generated

Figure 30- Definition of E1B08 from Experian (2024) documentation

**Scenario 2:** Some variables may have clear business names but the values used are codified and would not make sense without documentation to explain the values e.g. the AccountStatus variable shown below has 4 values but only shows 1 character. Some of these are intuitive but others can be confusing e.g. F standing for default.

A screenshot of a computer program

Description automatically generated

Figure 31 - Definition of AccountStatus from Experian (2024) documentation

These cases were going to be a key blocker to having an application that could effectively communicate its content to the end user if left unresolved. Functionaliy to provide easy to access definitions to the end user was required to avoid this issue. The nature of both issues meant that I could approach each separately.

#### Scenario 1 resolution

This was the more complicated scenario to solve, as the Experian API used as the data source contains 1000’s of variables per its documentation (Experian, 2024). Rendering a description for each of these did not seem practical or efficient.

The solution I arrived at to handle this was to instead provide the functionality for a user to automatically get the definition of a variable as needed. This presented an acceptable compromise between being able to communicate data to end users and keeping the application code simple.

To make this work, I needed a dictionary source that the application could use to look up variable descriptions. To keep this simple I decided to use a CSV file to store both the variable names and a description, the variable name being used to match to. Using this kind of lookup also allows the program to store it’s own definitions of variables if the documentation provided is lacking. For reading CSV files, I required a node.js library called csv-parser, so I installed this:

A screenshot of a computer program

Description automatically generated

Figure 32- Installing csv-parser

Using this library, I created a function within my Backend Node.JS express application. I did this in the Backend as trying to import one of the dependent node modules resulted in an error:

A screenshot of a computer

Description automatically generated

Figure 33- Error attempting to load a dependency directly into React for my lookup function

As I found out when trying to establish a connection to a SQL database, it’s easier to offload this logic to a dedicated Backend application rather than handle polyfills to get the logic working directly within the front-end application.

##### Backend Logic for lookup function

To provide a point of interaction, I needed to expose an Endpoint that the FrontEnd could send a HTTP call to. To do this, I created a new router object within the Node.js Express application, using one of the default ones provided by the officially provided generator (Express, n.d.) as a starting point. I added in a URL Parameter which will allow a variable name to be passed in to carry out a search for it’s definition. This is done via the “:searchKey” parameter seen in the router.get() method:

A screen shot of a computer program

Description automatically generated

Figure 34- Router.Get method for the backend to accept a search parameter

Once that endpoint is in place, the parameter is be passed into a dedicated asynchronous function that performs the lookup on the CSV file (the function is asynchronous as a regular function generated errors within the code editor). The function is then wrapped in a Try-Catch to account for the possibility of an error e.g. the input file not being available. The result is then loaded into a JSON payload and sent back the calling application (i.e. the FrontEnd) via the res.send() method.

A screen shot of a computer program

Description automatically generated

Figure 35 - Endpoint logic showing the lookup function being used & sending this back to the calling application

##### Front End Logic for lookup function

For the front-end, I coded the function as below:

A screen shot of a computer

Description automatically generated

Figure 36 - Code for Lookup functionality

This code will:

1. Accept an input parameter called “key”, which will hold the variable name we want a definition for
2. Attempt to retrieve the definition by using the “fetch” function to make a HTTP call to the Backend, using a host name stored within the process environment and passing in the “key” as part of the URL
3. Fetch will return 1 of 2 outcomes:
   1. An error is thrown if there is a network error
   2. Parse a successful response for a variable called “description”
4. Assuming a successful response, the description returned by the Backend is provided to the user via Javascript’s inbuilt “alert” method

Each variable that contains data the end user can review is rendered in the application by a React component called “KeyValueRow”. Within this component, this lookup function is then called via the “onClick” property of the table row that displays the data to the user:A screen shot of a computer program

Description automatically generated

Figure 37- OnClick property using the lookup function

When the function is successful, a definition will pop up on the end users screen e.g. for the E1B08 variable, the below prompt is currently presented to the user. Below is an example of this done in the Firefox web browser:

A screenshot of a computer

Description automatically generated

Figure 38 - Lookup prompt presented to the user for E1B08 as seen in Firefox

Below is another example of this functionality done in the Microsoft Edge web browser:

A screenshot of a computer

Description automatically generated

Figure 39- Lookup prompt presented to the user for E1B08 as seen in Microsoft Edge

The row within the table will also highlight, to give the user an indication that the row can be clicked on to bring this definition up. An instruction is also given at the top of the table.

Using an alert textbox for this allows the description to be communicated to the end user easily without the need for more complex HTML & CSS components to render the description on page.

#### Scenario 2 resolution

Translating the values of a given variable into business friendly language could be done in a more simple approach then the 1st scenario. So long as the documentation provided a dictionary of values, it was possible to code a switch statement to return a detailed value that communicated more clearly with business users. Taking the “AccountStatus” variable as an example, I added this switch case into the code:

A screen shot of a computer

Description automatically generated

Figure 40- Switch case to translate AccountStatus into a business friendly value

This is written directly into a React component called “CaisRow”, which receives the coded value for AccountStatus as part of a prop called “CAISDetails”. This is then parsed to get the raw value. A business friendly value can then be returned by simply passing this value into the switch case:

A screen shot of a computer program

Description automatically generated

Figure 41 - CaisRow component return statement, showing the translated value in use

This “CaisRow” component gets rendered multiple times based on the data received by the program:

A screen shot of a computer program

Description automatically generated

Figure 42- React JSX code showing the rendering of the CaisRow components via the .map() method

When the React application renders these CaisRow components, we can see a clear test value has been rendered for the user, instead of the shortened value that could be mis-interpreted.

A screenshot of a computer

Description automatically generated

Figure 43 - Rendered React application showing the business friendly language used instead of the original values

This same technique was then applied to any similar variables, allowing clear values to be shown instead of original values requiring documentation to interpret.

### Environments configuration

One stretch goal I wanted to achieve with this project was to have the final application in a state where it would be ready to be deployed to an actual environment aside from the localhost I developed on. The aspiration was to have a UAT version of the application I could share with colleagues and gather feedback on the usefulness of the app & how easy it was to navigate.

To do this though, I had to deal with a logical flaw in my code. For example, to meet the requirement of displaying a list of recent applications, the front end React app makes a HTTP call to the Backend via the “fetch” method, which enables HTTP calls to be made in base JavaScript (W3Schools, 2024, Fetch API). In my first version of this however, I hardcoded the URL the fetch method would call, as seen below:

A screenshot of a computer program

Description automatically generated

Figure 44 - Hardcoded URL within the React Application

This logically would not work anywhere but my local machine, so this was a barrier to meeting this deployment aspiration. To address this, I created a “.env” file that my React App could read from when it is started to get the appropriate variables for a given environment:

A screenshot of a computer

Description automatically generated

Figure 45 - .env file added to allow dynamic configurations without altering the raw code

I then altered my code to use this environment instead of the hard-coded value it previously had:

A screen shot of a computer screen

Description automatically generated

Figure 46 - Altered code to use the newly created .env file

For localhost development, I used the same localhost:port URL I had before, allowing the app to still work on my development machine. When it comes to a deployment however, I can now swap this .env file for a configuration appropriate to the environment e.g. a ”.env.uat” file with the URL for a UAT version of the back end.

The naming of the variable is due to React requiring the “REACT\_APP\_” prefix on all variables, which allows React to find the variables when using the default “npm start” command to start the server (Facebook, 2024).

# Final Tech Stack

By the end of my project, the final Tech Stack looked like this:

A black and white screen with white text

Description automatically generated

Figure 47 - Diagram showing the Tech Stack of the full application

# Feedback - Review with Employer

When I was ready to show the progress made on this, I made use of a regular 1 to 1 with my Line Manager to gather some feedback on the application.

A screenshot of a computer

Description automatically generated

Figure 48 - Teams Call arranged to discuss Apprenticeship progress

I showed the React web pages I had created and explained how the functionality works. Below are minutes of the feedback received:

* *Overall, impressed with the work completed so far*
* *Navigating the web application was simple and effective*
* *The Summary and Premium Value pages are effective in how they can render all the data points with minimum code*
* *The Lookup function has good potential, however the CSV file used may not age well given the complexity of the data the application is using. Consider an alternative*

One concern I did raise was the timescales it would take to render all the data provided by Experian (2024) on the RawData page of my application. Given the scope of the project as is, my Line Manager advised that I should focus only on the most used data blocks provided by Experian. We discussed that the CAIS (Credit Account Information Sharing) was by far the most useful data block that could be shown here, so I prioritised rendering this data on the RawData page.

# Final Application Pages

The below figures show what the final web pages look like.

## Home.jsx

A screenshot of a computer

Description automatically generated

Figure 49- Home Page (final)

## SummaryData.jsx

A screenshot of a computer

Description automatically generated

Figure 50 - Summary Data page (final)

## PremiumValue.jsx

**A screenshot of a computer

Description automatically generated**

Figure 51- Premium Value page (final)

## RawData.jsx

A screenshot of a computer

Description automatically generated

Figure 52- Raw Data page (final)

# Conclusions

Below are my thoughts from reflecting on the work completed for this project:

## Project Outcome

For the most part, the final application did achieve it’s aims and met its requirements, however I did run into many challenges being able to make this work.

Being unable to include the “mssql” Node.js package directly into my React application was a significant hurdle I had to overcome to complete this project, as making a database connection was a fundamental requirement set in the project acceptance criteria. Researching the errors I was getting led me to concepts I had not encountered before e.g. polyfills. I was able to eventually resolve the errors I faced by setting up a dedicated backend, but this had the impact of unexpected scope creep.

## Future Additions

If I were to take this project further, below are a few things I would like to add/do differently:

* Try using a different framework for the backend rather than the Node.js Express framework. I found this framework to be awkward to use for the functionality I needed, mainly due to the confusing syntax around how it defines it’s endpoints and how it makes the request and response available to the developer
* Explore using a dedicated CSS Framework to handle the formatting of the Front End application. CSS is something I have not been enjoying working with, so Frameworks to simplify its use would be preferable to use in future. Frameworks I could look at include: Bootstrap, Tailwind & Foundation (W3Techs, 2024 and WeAreDevelopers, 2023)
* Rethink how the lookup function in my code used to display the data definitions works. While the version I have works, it has a few flaws I’m not satisfied with:
  1. Using the native Javascript alert() function poses a breaking issue if the users machine blocks these e.g. Firefox provides this functionality (Mozilla Support, n.d.). It would be preferable to display a dedicated HTML element with the information that can be dismissed by the user.
  2. A CSV file I don’t feel is the most reliable way to store this information. One key issue I have is that the current solution only works reliably if there is a single record in the file to return, which I believe would break from a full UAT test of the application. From speaking to a peer privately (who does not work for Vanquis), an alternative we identified was a SQLite database, which is a lightweight public domain database what can be deployed directly with an application (SQLite, n.d.).
* Discover how to get a working deployed version of the full application (Front End and Back End) that are capable of communicating with each other. The furthest I got with this project was deploying a Front End only (see Appendix). This will be critical to making my developed applications available to end users.

# References

Amazon AWS. (2024). What is SDLC (Software Development Lifecycle)?. [Online]. AWS. Last Updated: 2024. Available at: <https://aws.amazon.com/what-is/sdlc/#:~:text=The%20software%20development%20lifecycle%20(SDLC,expectations%20during%20production%20and%20beyond>. [Accessed 28 June 2024].

Barger, R. (2021). How to Create a React App with a Node Backend: The Complete Guide. [Online]. FreeCodeCamp. Last Updated: 2021. Available at: https://www.freecodecamp.org/news/how-to-create-a-react-app-with-a-node-backend-the-complete-guide/ [Accessed 5 July 2024].

Experian. (2024). DelphiSelect API Documentation - for Credit Decisioning data. [Online]. DelphiSelect API Documentation. Last Updated: 2024. Available at: <https://developer.experian.com/products/uk/delphi-select/delphi-select-api-docs> [Accessed 17 May 2024].

Express. (n.d.). Express application generator. [Accessed July 6, 2024], from <https://expressjs.com/en/starter/generator.html>

Facebook. (2024). Adding custom environment variables. *Create React App Documentation*. Available at: <https://create-react-app.dev/docs/adding-custom-environment-variables/> (Accessed: 5 August 2024).

Iconic Digital Marketing Agency. (2024) The importance of brand guidelines. Available at: <https://iconicdigitalagency.com/blog/branding/the-importance-of-brand-guidelines/> (Accessed: 5 August 2024).

Mozilla. (2024a). Polyfill. [Online]. developer.mozilla.org. Last Updated: 2024. Available at: <https://developer.mozilla.org/en-US/docs/Glossary/Polyfill> [Accessed 5 July 2024].

Mozilla. (2024b). *CSS - Top Property*. [Online]. Mozilla Developer. Last Updated: 2024. Available at: https://developer.mozilla.org/en-US/docs/Web/CSS/top [Accessed 9 August 2024].

Mozilla Support. (n.d.). Pop-up blocker settings, exceptions and troubleshooting. [online] Available at: <https://support.mozilla.org/en-US/kb/pop-blocker-settings-exceptions-troubleshooting> [Accessed 6 Aug. 2024].

Microsoft. (2024). *Proof of concept connecting to SQL using Node.js*. [Online]. . Available at: [Step 3: Connecting to SQL using Node.js - Node.js driver for SQL Server | Microsoft Learn](https://learn.microsoft.com/en-us/sql/connect/node-js/step-3-proof-of-concept-connecting-to-sql-using-node-js?view=sql-server-ver16) [Accessed 9 August 2024].

React Router. (n.d.). Link. [Online]. React Router - Link Object. Available at: <https://reactrouter.com/en/main/components/link> [Accessed 9 August 2024].

SQLite. (n.d.). SQLite Features. [online] Available at: <https://www.sqlite.org/features.html> [Accessed 6 Aug. 2024].

Stack Overflow. (2014). No 'Access-Control-Allow-Origin' - Node / Apache Port Issue. [Online]. Stack Overflow. Last Updated: 2014. Available at: <https://stackoverflow.com/questions/18310394/no-access-control-allow-origin-node-apache-port-issue> [Accessed 9 August 2024].

Stack Overflow. (2024). babel-preset-react-app, is importing the "@babel/plugin-proposal-private-property-in-object" package without declaring. [Online]. Stack Overflow. Last Updated: 2024. Available at: <https://stackoverflow.com/questions/76435306/babel-preset-react-app-is-importing-the-babel-plugin-proposal-private-propert> [Accessed 28 June 2024].

Yasar, K. (2024) 'What is an API endpoint?', *TechTarget*, July. Available at: <https://www.techtarget.com/searchapparchitecture/definition/API-endpoint> (Accessed: 5 August 2024).

WeAreDevelopers. (2023). Best CSS Frameworks. [online] Available at: <https://www.wearedevelopers.com/magazine/best-css-frameworks> [Accessed 6 Aug. 2024].

W3Schools. (2024). Fetch API. Available at: <https://www.w3schools.com/jsref/api_fetch.asp> (Accessed: 5 August 2024).

W3Schools – CSS Position. (2024). *CSS Layout - The position Property*. [Online]. W3Schools - position. Last Updated: 2024. Available at: <https://www.w3schools.com/css/css_positioning.asp> [Accessed 9 August 2024].

W3Techs. (2024). Usage statistics and market share of CSS frameworks for websites. [online] Available at: <https://w3techs.com/technologies/overview/css_framework> [Accessed 6 Aug. 2024].

# Appendix

Below is some additional work I did for this project, but have not included as part of the final project. Reasons for this include:

* Work was made redundant with later commits made to the repositories
* Work was incompatible with the end goal of the project or it’s asperations

## Initial Backend attempt, with React & Node.JS using concurrently

The “mssql” module I need to connect my React application to an Azure SQL Database is a Node.js module, so I needed to create a Node Backend to be able to make use of this. I follows I guide from (Barger, R, 2021) to do this.

For this to work, I required the “express” module, so used npm to ensure that was available:A screen shot of a computer program

Description automatically generated

Figure 53- Initial Backend npm start logic

This worked on my local machine. However it did not when I deployed to my Azure host. Rather then debug this, I thought it would be easier to implement a separate Node.js server as a backend and access as an HTTP call in the React Application. Therefore, this was scrapped.

## Hosting in Azure Static Web App

One thing I wanted to do to allow testing of the application is to make it available to other users, which would enable UAT testing directly with multiple colleagues. To do this though, I needed a way to deploy the application.

I already had a Visual Studio Professional subscription (which has monthly Azure credits) from my employer, so this was my 1st idea for a hosting solution. I came across the Static Web App resources, which would allow me to link to the GitHub repository directly. Shown below is how I configured this:

A screenshot of a computer

Description automatically generated

Figure 54 - Setting up an Azure Static Web App to host the React application

A screenshot of a web application

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

After the deployment, I tried using the default URL Azure provided to see if I could access the application. This was successful:

A screen shot of a computer

Description automatically generated

Figure 55 - The React App home page as deployed to the Azure Static Web App

### Confirming access

I needed to know if this deployed application was accessible by other users. To confirm this I asked one of my colleagues (referred to as MO in the screenshot below) to try accessing the link and confirm what they see. This was successful:

A screenshot of a chat

Description automatically generated

Figure 56 - Confirmation a user can access the deployed React App

This gave me a proof of concept that this method of deploying a React Application to an Azure based host would work with the resources available to me. When the need for a full Backend became apparent however, I needed to consider how the deployed application would communicate with the Backend, which I did not consider with this Static Web App. I therefore scrapped this and investigated alternatives.