**Carer’s Allowance SMS project (CASMS)**

**Alpha Mackie**

**Software Developer Level 4**

Introduction

The Department for Work and Pension (DWP) was experiencing a large volume of calls from customers who have submitted a claim for Carer’s Allowance and were waiting for their first payment. Therefore, this initiative was introduced to free up time for service centre agents to focus on more value-added tasks. Previously agents had to navigate multiple screens to get customer information and manually input data on other screens, which increased the risk of human error when sending SMS messages to customers. In order to mitigate this, the repetitive tasks were automated.

The technologies used for this project are:

* JavaScript, which is the main programming language used.
* Embedded JavaScript (EJS), the templating language to generate data dynamically from the database to the application pages.
* Mongo DB Compass for the database.
* Jira, the tool used for managing and tracking the work done by my team.
* The Integrated Development Environment (IDE), Visual Studio Code (VS Code) was used to rebuild the pages with HTML and CSS.

The automation processes my team built has to interact with three systems known as CAMLite, Reach API and Carer’s Allowance Customer System (CACS).

My contribution to this automation process was, building the dummy Database that CACS interacts with, Reach API calls using the customer’s mobile number and the CAMLite automation Worker Robot.

This submission covers the repairing of one of the CACS screens, the creation of the dummy database and fetching data from the database; and displaying on screen.

In terms of user stories, I was assigned the user story by one of the senior devs in my team whose expertise is UI and Databases. This user story was only assigned to me so that I can gain some more exposure to repairing HTML pages; on the user story the screen is shown as ICA400400, but it is the same screen as ICA400404. I was not assigned a user story for coding the database functionality, just verbal instructions. Please see the user story below:

A screenshot of a cell phone

Description automatically generated

*Figure1: User story for ICA400404 which is the same as ICA400400*

Rebuilding CACS screen (ICA400404)

Screen ICA400404 is one of the screens in the CACS application that the software Robots we built have to navigate through. It is a static page that displays a customer’s case summary information. In the development environment, we don’t use the actual CACS application, we have to mimic the way the Robot will navigate the various screens to collect and input data during the automation process. Please see below the screen that I had to rebuild:

A screenshot of a social media post

Description automatically generated

*Figure 2: The corrupt Static HTML page(screen) ica400404*

Below is the same page after I have rebuilt it:

A screenshot of a cell phone screen with text

Description automatically generated

*Figure 3: The finished ica400404 screen.*

The IDE used is Visual Studio Code (VS Code) as required by our IAG standards. The original HTML code was not written by me, I had to make some adjustments by changing the values for the borders and paddings and margins. My goal was not to re-invent the wheel but to fix the existing code in order to get the screen to look like the actual application screen. The rebuilding of the page involves repositioning the heading, text and input fields to look exactly like the CACS application screens as seen in the image above. It is important to note that I did not write all the html code from scratch. I formatted the code, used inline styling to adjust the borders, paddings and margins as this is what was used in the original pages. As seen above on figure 3, there is plain border styling for the input fields, a black background colour and white text. I also used the ‘<pre></pre>’ tags with preformatted text so that the text within the tags preserve both spaces and line breaks where necessary. The input tags with inline styling and embedded ‘ejs’ (example code- <%=data.nino%>) code to display the data from the database dynamically. The code in bracket gets the national insurance number from MongoDB and displays it on the screen. Please see the HTML code embedded with ‘ejs’ code below:

A close up of a screen

Description automatically generated

*Figure 4a: The raw HTML showing the inline css styling and EJS code.*

A screen shot of a computer

Description automatically generated

*Figure 4b: The raw HTML showing the inline css styling and EJS code.*

I also used input tags with inline styling and HTML event attributes like keypress, mouseover, keyUp, onFocus and onClick for on screen actions. The rest of the other lines of code was pre-written by the original screen builder.

Database in MongoDB Compass

I created the database called ‘cacs-db’ with dummy data in MongoDB, to reflect the information that the application requires. MongoDB stores documents as a collection and to add a collection to the database, the simplest way to do this is, you use the cursor to hover over the ‘cacs-db’ which is the database, a plus sign will appear, which you then click. This will add an empty collection with the ID auto generated by default but can be change. The data can be entered manually or the green icon with ‘add data’ displayed, can be clicked on, to import data from a file. In each collection, I included two fields; the ‘Nino’ which is the national insurance number and ‘OPSD’ which is the first payment date, that will be displayed on screen ica400404.

Please see image below:

A screenshot of a cell phone

Description automatically generated

*Figure 5: MongoDB Compass with data for screen ica400404.*

Getting the data from MongoDB and displaying on screen.

For getting the data from MongoDB and displaying on screen, I took three steps:

1.Creating a schema.

2.Establishing a database connection

3. Querying the database.

Creating the schema

The IAG has adopted the Model-View-Controller (MVC) architectural pattern which I followed. The Model folder has all the files for the database schemas, the View contains all the application screens and in the Controller folder you will find all the files with code for getting and displaying data on screen from MongoDB. There is also a folder called ‘routes’, in it you will find all the files with code that gets you to the screens and connections to the database.

As with the screen creation, the IDE environment was VS Code and the Programming language was JavaScript and a templating language known as ‘EJS’.

Firstly, I wrote the schema code for screen ICA400404 in which I imported Mongoose from the Mongoose module, installed using the Node Package Manager (NPM) command on the command line. Secondly, I defined a new variable called Schema which would ref to the Schema object in the Mongoose module. I then initialise a new schema with an ‘\_id’ of object type id, which is created automatically by MongoDB, and two other items of type string, as reflected in the MongoDB database. I finally export the module with the mongoose model, passing the collection name ‘ICA400404’ and the new schema ‘CustomerSchemaOPSD’. Please see image below:

A screenshot of a computer

Description automatically generated

*Figure 6: The schema for screen ica400404*

Establishing a database connection

To establish a connection, I first created a sub folder called ica400404 in the routes folder and a file I named index.js. This file contains the code to establish connection to MongoDB using Mongoose and a callback function logging a message that the connection was successful for the collection or table ’ica400404’ in the database. I then imported the Router object from express and add a route using the get method, and a function called ‘read’ that reads the data. Lastly, I exported the router object.

Please see the image below:

A screenshot of a cell phone

Description automatically generated

*Figure 7: The code to establish the DB connection*

*A screenshot of a cell phone

Description automatically generated*

*Figure 8: Database connection established*

Querying the database

For Querying the database, I created a sub folder in the ‘controllers’ folder with the same name as the sub folder in the routes folder. I then required the ‘CustomerSchemaOPSD’ object from ‘models’ folder. I used the read function to get the data with the ‘nino’(national insurance number), apply the ‘findOne’ method in the ‘CustomerSchemaOPSD’ object with the ‘nino’ object and a callback function passing error and data, checking if there is an error, to throw an error message and if the data is not null; apply the render method to render the page, otherwise return to the ‘ese045’ page or screen and display the message that the page was not found. I decided to use this feature of JavaScript, the callback function to get feedback in the form of an error message if and when an error occurs or render the page if there is data. My other option would have been to use the ‘promise’ or ‘async and await’ feature.

Please see the image below: A screenshot of a cell phone

Description automatically generated

*Figure 9: The code to query the database*

Conclusion

The automation part of this project was done in Uipath Studio for which I was assigned user stories. However, for creating the mock application, the user stories were assigned to my senior colleagues and I was asked to work on the above just for me to gain some more experience in databases and Ui development as well as connectivity. Fixing the UI enabled my team to write the UIpath code that mimics the actions of a human user to input data to query a database. The dummy database was used to store and retrieve data just like in the actual application. I was given verbal instructions by my senior colleague in the team, who is an expert in that area and this has boosted my confidence in handling some backend development. Furthermore, being able to present my work in this submission has reinforced my understanding of the subject matter. Apart from using the Internet Explorer developer tools for debugging the web pages, the command line to test the code I wrote in VS code, the unit test was done by a dedicated tester. The mock database was used to mimic the real application when running the automation. When the unit testing was done, the page I repaired loaded well and the database connection worked successfully with the data displayed on the page as it would in the actual applications.