Criterion C: Development

Techniques Used

1 Database Interactions

Given the client's requirement that the transactions be accessible in both of his primary devices (Criterion 9), local database solutions (like SQLite, localStorage) were ruled out. Therefore, the use of an online database was considered to be the most appropriate.

For this purpose, Firebase and Firestore (Firestore being a NoSQL database introduced and provided by Firebase) were used in the product as their services were used to effectively fulfil many of the criteria:

- User authentication and application security (Criterion 2) and signing in feature using Gmail accounts (Criterion 1)
- Adding, storing (Criterion 3) and managing (Criterion 5) transactions
- Storing and managing transaction categories (Criterion 6)
- Being able to access transactions on both of the client's primary devices (Criterion 9)

Firestore's file-based storage system (shown in Figure 1) facilitates complex querying and high storage scalability for free *(GeeksForGeeks)*, which enabled features such as generating graphs for the Reports page (Criterion 8).

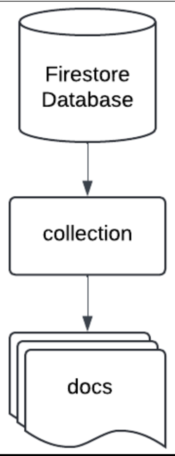
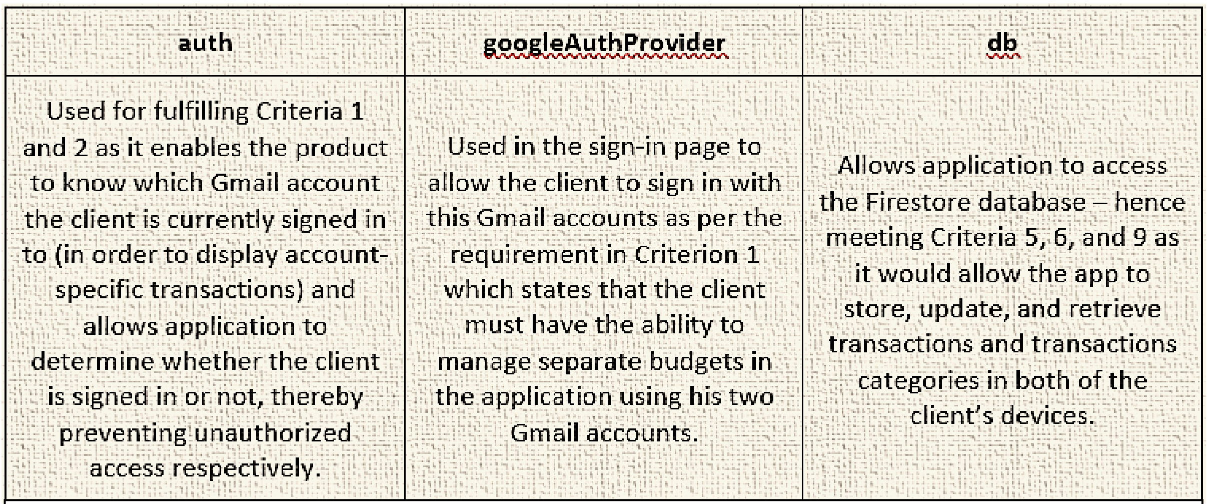


Figure 1: Firestore stores each transaction and category as individual 'docs'. These are organized into separate groups called 'collections' (as exemplified in Figure 6 below).

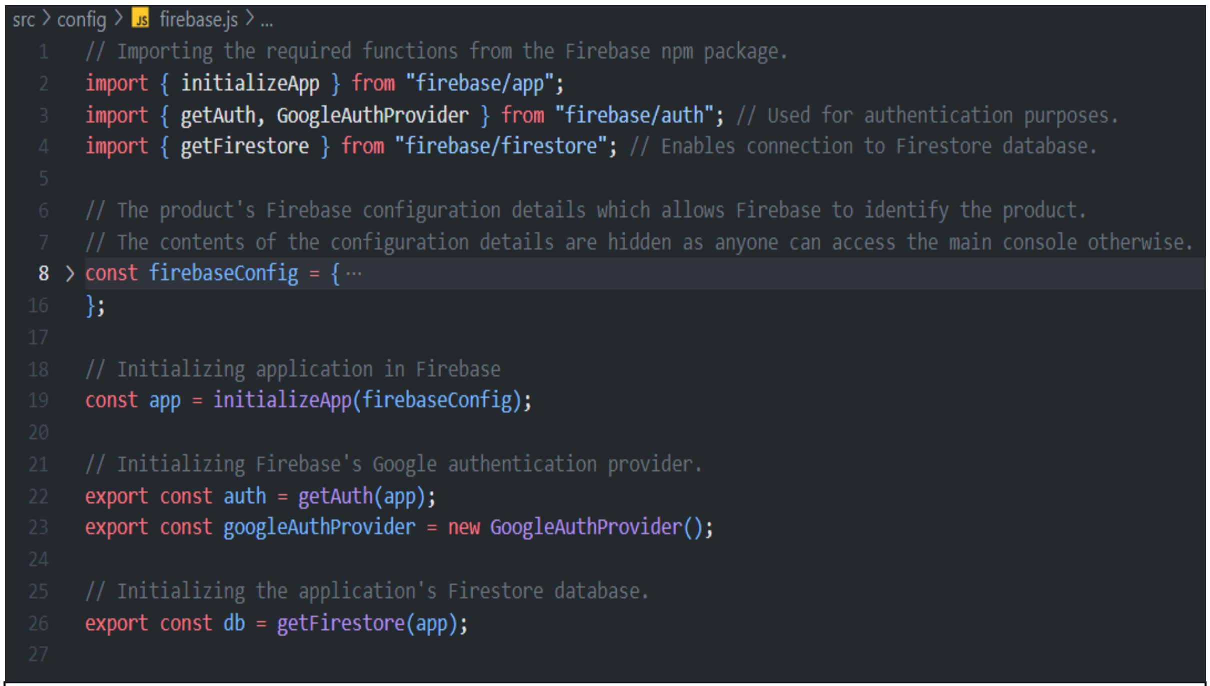
1.1 Firebase Connection

The product imports key Firebase and Firestore functions and services through the *firebase.js* configuration file (Figure 3). The functions and services are imported through the Firebase npm package *(Firebase)*. Firestore and Firebases' features that are used throughout the application are detailed in Figure 2 below:



auth	googleAuthProvider	db
Used for fulfilling Criteria 1 and 2 as it enables the product to know which Gmail account the client is currently signed in to (in order to display account-specific transactions) and allows application to determine whether the client is signed in or not, thereby preventing unauthorized access respectively.	Used in the sign-in page to allow the client to sign in with this Gmail accounts as per the requirement in Criterion 1 which states that the client must have the ability to manage separate budgets in the application using his two Gmail accounts.	Allows application to access the Firestore database – hence meeting Criteria 5, 6, and 9 as it would allow the app to store, update, and retrieve transactions and transactions categories in both of the client's devices.

Figure 2: Important functions imported from the Firebase npm package (Firebase) and/or defined in the firebase.js configuration file (Figure 3, Lines 3, 23, 26).

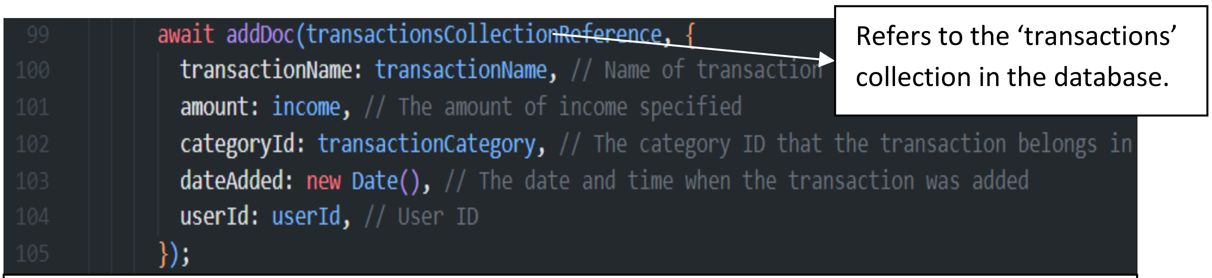


src > config > firebase.js > ...

```
1  // Importing the required functions from the Firebase npm package.
2  import { initializeApp } from "firebase/app";
3  import { getAuth, GoogleAuthProvider } from "firebase/auth"; // Used for authentication purposes.
4  import { getFirestore } from "firebase/firestore"; // Enables connection to Firestore database.
5
6  // The product's Firebase configuration details which allows Firebase to identify the product.
7  // The contents of the configuration details are hidden as anyone can access the main console otherwise.
8 > const firebaseConfig = {···
16 };
17
18 // Initializing application in Firebase
19 const app = initializeApp(firebaseConfig);
20
21 // Initializing Firebase's Google authentication provider.
22 export const auth = getAuth(app);
23 export const googleAuthProvider = new GoogleAuthProvider();
24
25 // Initializing the application's Firestore database.
26 export const db = getFirestore(app);
27
```

Figure 3: firebase.js configuration file.

1.2 Adding Transactions – IncomeForm Component

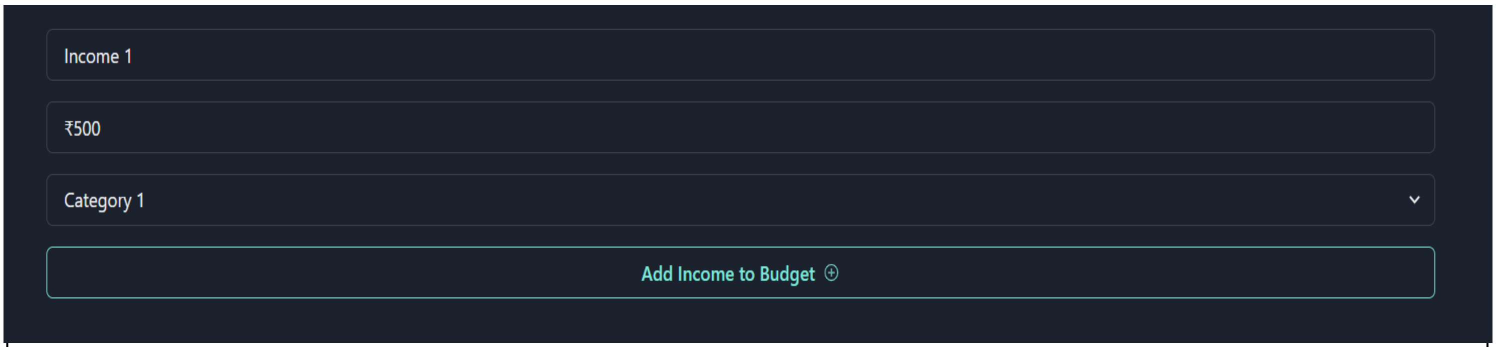


```
99      await addDoc(transactionsCollectionReference, {
100       transactionName: transactionName, // Name of transaction
101       amount: income, // The amount of income specified
102       categoryId: transactionCategory, // The category ID that the transaction belongs in
103       dateAdded: new Date(), // The date and time when the transaction was added
104       userId: userId, // User ID
105     });
```

Refers to the 'transactions' collection in the database.

Figure 4: Lines 99-105 from the IncomeForm.js component file.

There are two types of transactions that the client can add – incomes and expenses – which can be added through the IncomeForm and ExpenseForm components respectively. Both components directly fulfil Criterion 3 by enabling the client to add new incomes and expenses in the home page as shown in Figure 5. The *addDoc* function provided by Firestore *(Firebase)* is used for this function (Figure 4, Line 99) as it sends the to-be-created transaction doc's field values (Figure 4, Lines 100-104) from the application frontend to the Firestore database to be stored in the 'transactions' collection as a new doc (sample income doc shown below in Figure 6). Since the *addDoc* function is asynchronous, it provides extra functionality to the client, as elaborated upon in Section 2.

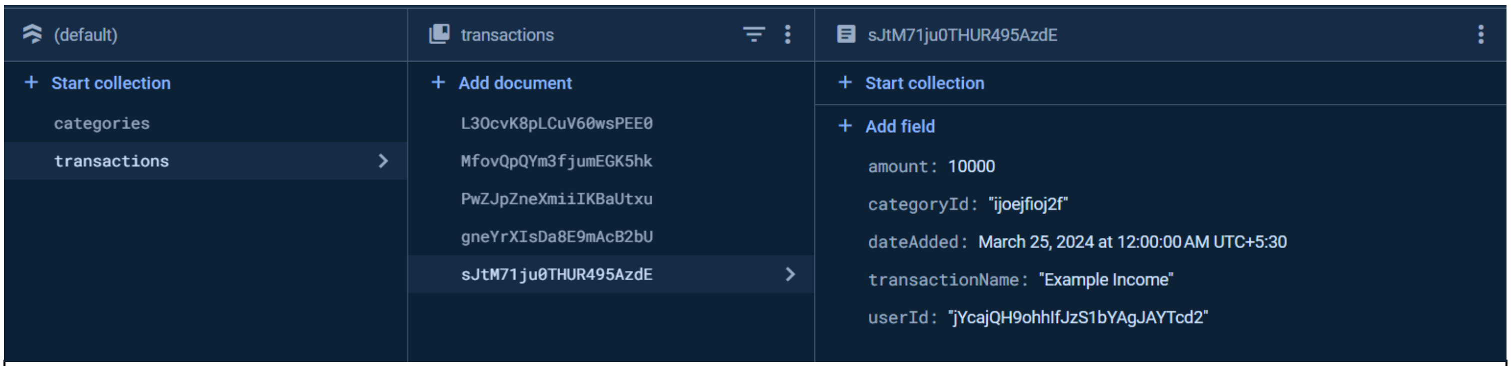


Income 1

₹500

Category 1

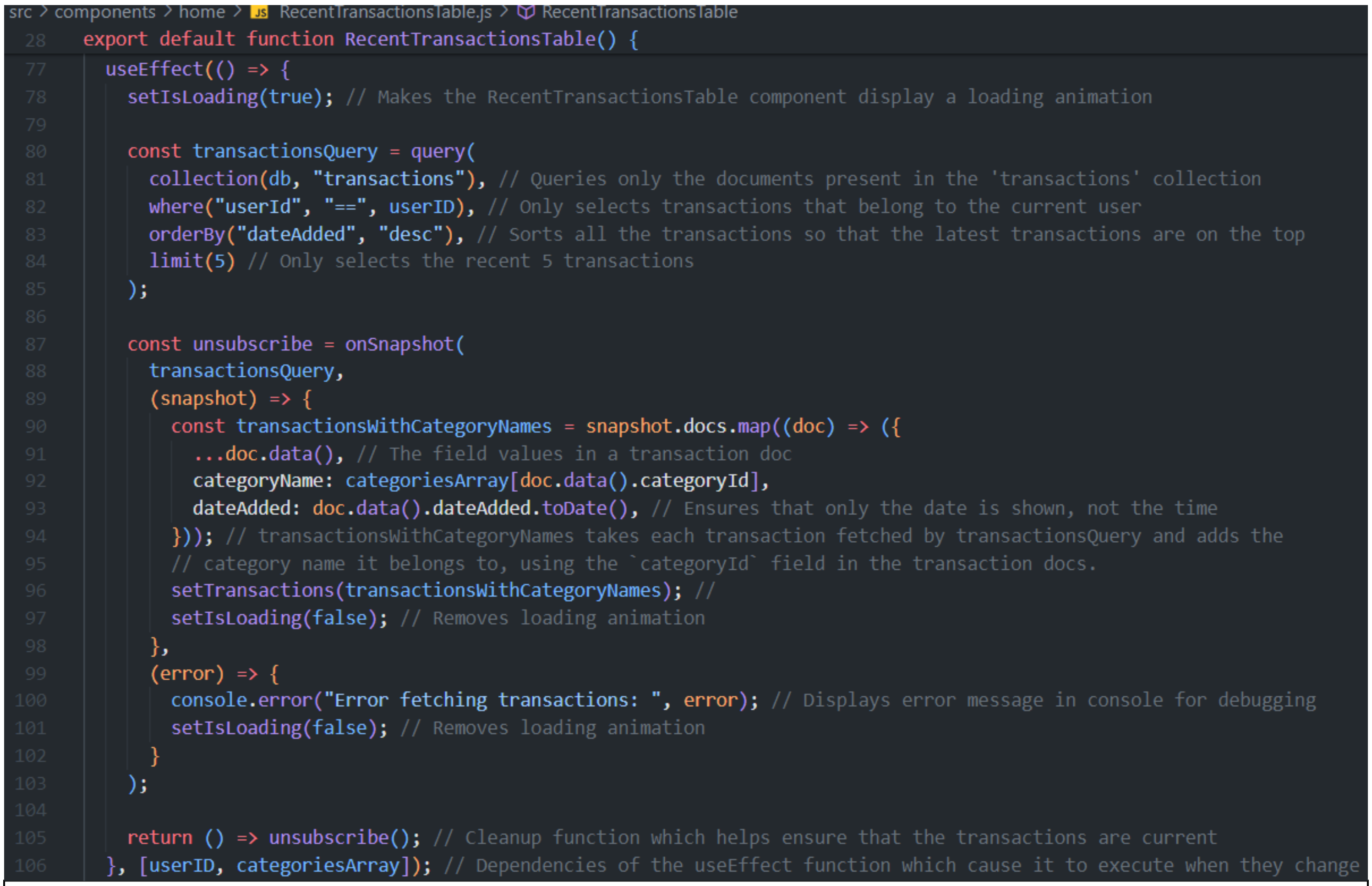
Figure 5: Screen capture of the IncomeForm component with filled-in sample transaction data.



amount: 10000
categoryId: "ijoejfioj2f"
dateAdded: March 25, 2024 at 12:00:00 AM UTC+5:30
transactionName: "Example Income"
userId: "jYcajQH9ohhIfJzS1bYAgJAYTcd2"

Figure 6: Screenshot of the structure of the Firestore database. The field values of a sample transaction doc from the 'transactions' collection are shown.

1.3 Fetching Transactions - RecentTransactionsTable Component



```
src > components > home > RecentTransactionsTable.js > RecentTransactionsTable
 28  export default function RecentTransactionsTable() {
 77    useEffect(() => {
 78      setIsLoading(true); // Makes the RecentTransactionsTable component display a loading animation
 79
 80      const transactionsQuery = query(
 81        collection(db, "transactions"), // Queries only the documents present in the 'transactions' collection
 82        where("userId", "==", userID), // Only selects transactions that belong to the current user
 83        orderBy("dateAdded", "desc"), // Sorts all the transactions so that the latest transactions are on the top
 84        limit(5) // Only selects the recent 5 transactions
 85      );
 86
 87      const unsubscribe = onSnapshot(
 88        transactionsQuery,
 89        (snapshot) => {
 90          const transactionsWithCategoryNames = snapshot.docs.map((doc) => ({
 91            ...doc.data(), // The field values in a transaction doc
 92            categoryName: categoriesArray[doc.data().categoryId],
 93            dateAdded: doc.data().dateAdded.toDate(), // Ensures that only the date is shown, not the time
 94          })); // transactionsWithCategoryNames takes each transaction fetched by transactionsQuery and adds the
 95          // category name it belongs to, using the `categoryId` field in the transaction docs.
 96          setTransactions(transactionsWithCategoryNames); //
 97          setIsLoading(false); // Removes loading animation
 98        },
 99        (error) => {
100          console.error("Error fetching transactions: ", error); // Displays error message in console for debugging
101          setIsLoading(false); // Removes loading animation
102        }
103      );
104
105      return () => unsubscribe(); // Cleanup function which helps ensure that the transactions are current
106    }, [userID, categoriesArray]); // Dependencies of the useEffect function which cause it to execute when they change
```

Figure 7: Lines 77-106 from the RecentTransactionsTable.js component file.

The RecentTransactionsTable component displays the five most recent transactions created by the client in the Home page, which was a problem for the client in his previous setup using Excel (Appendix A1), which also directly meets the requirements set in Criterion 4.

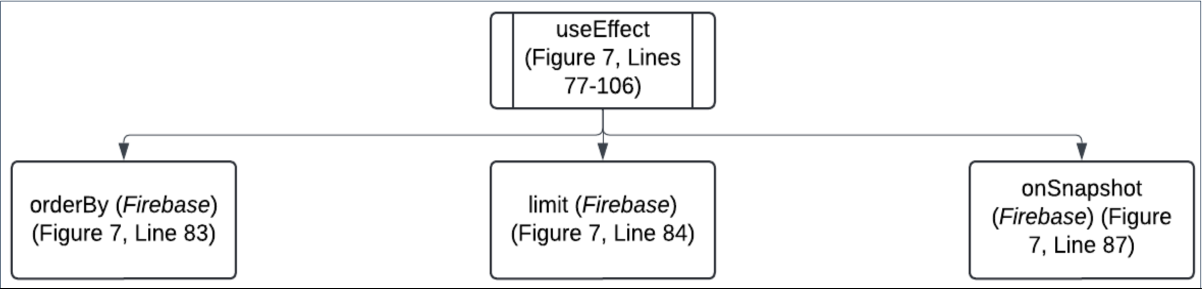
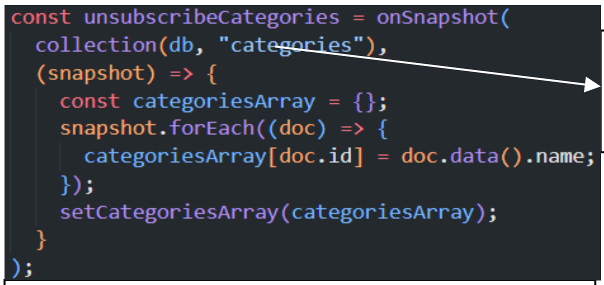

useEffect (Figure 7, Lines 77-106)
orderBy (Firebase) (Figure 7, Line 83)
limit (Firebase) (Figure 7, Line 84)
onSnapshot (Firebase) (Figure 7, Line 87)

Figure 8: Process flowchart based on code in Figure 7. useEffect function allows for retrieval of transactions from the database to initiate as early as possible, displaying recent transactions faster and more effectively meet Criterion 4.

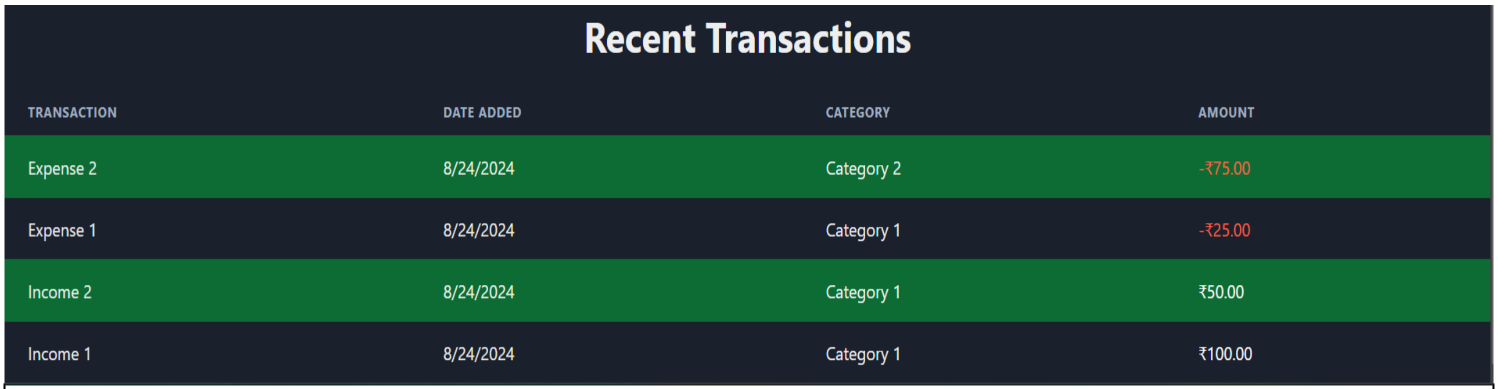


```
const unsubscribeCategories = onSnapshot(
  collection(db, "categories"),
  (snapshot) => {
    const categoriesArray = {};
    snapshot.forEach((doc) => {
      categoriesArray[doc.id] = doc.data().name;
    });
    setCategoriesArray(categoriesArray);
  }
);
```

Refers to the 'categories' collection in the database.

Figure 9: Lines 56-65 from the RecentTransactionsTable.js component file.

However, the RecentTransactionsTable component needs to obtain the category name associated to the categoryId of each transaction (since each transaction doc only stores the ID of the category to which it belongs (Figure 6)). It is not feasible for the client if he does not know the name of the category which each transaction belongs to. Hence, using the categoryId, the category's name is fetched from the categoriesArray (Figure 9, Lines 56-65) and displayed in the RecentTransactionsTable (Figure 10), thus contributing to the fulfilment of Criterion 4 since each transaction record can also display the name of the category that the transaction belongs to – hence providing the client more detailed information about his recent transactions.

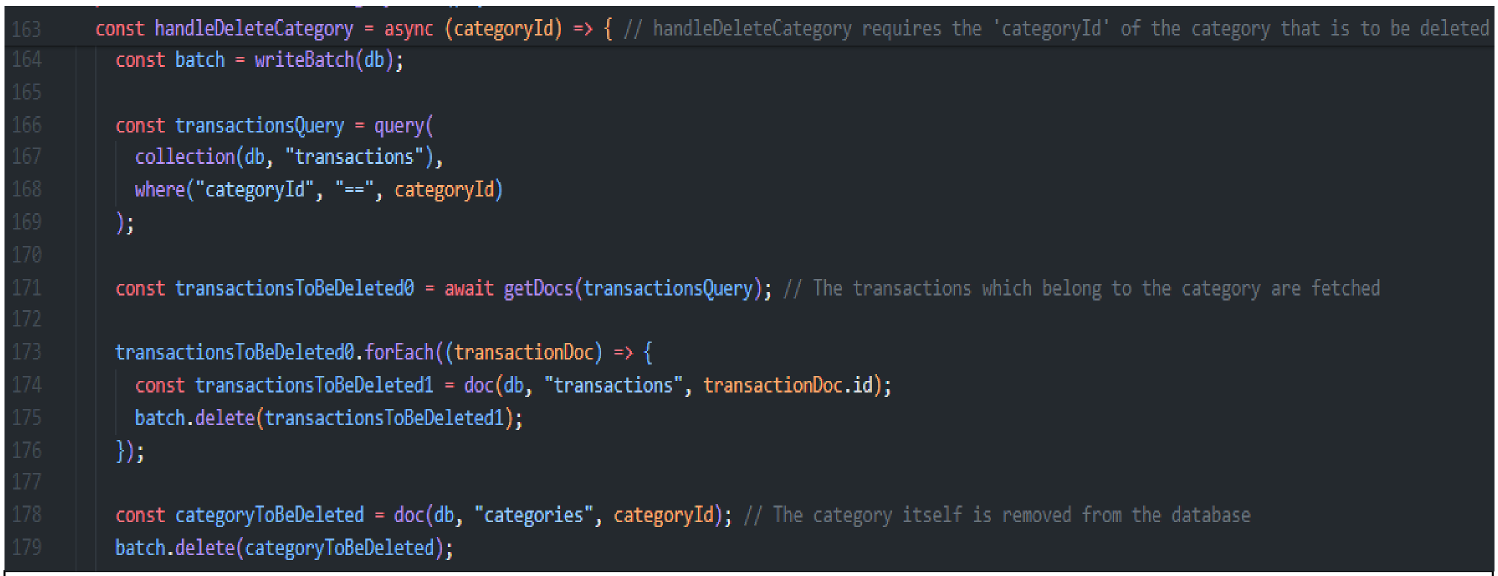


Recent Transactions

TRANSACTION	DATE ADDED	CATEGORY	AMOUNT
Expense 2	8/24/2024	Category 2	-₹75.00
Expense 1	8/24/2024	Category 1	-₹25.00
Income 2	8/24/2024	Category 1	₹50.00
Income 1	8/24/2024	Category 1	₹100.00

Figure 10: Screen capture of the RecentTransactionsTable component with sample transactions.

1.4 Deleting Categories - CategoryTable Component



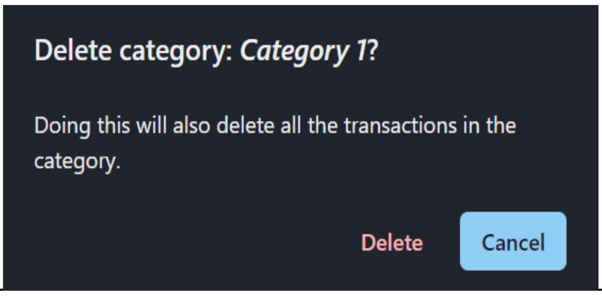
```
163  const handleDeleteCategory = async (categoryId) => { // handleDeleteCategory requires the 'categoryId' of the category that is to be deleted
164    const batch = writeBatch(db);
165
166    const transactionsQuery = query(
167      collection(db, "transactions"),
168      where("categoryId", "==", categoryId)
169    );
170
171    const transactionsToBeDeleted0 = await getDocs(transactionsQuery); // The transactions which belong to the category are fetched
172
173    transactionsToBeDeleted0.forEach((transactionDoc) => {
174      const transactionsToBeDeleted1 = doc(db, "transactions", transactionDoc.id);
175      batch.delete(transactionsToBeDeleted1);
176    });
177
178    const categoryToBeDeleted = doc(db, "categories", categoryId); // The category itself is removed from the database
179    batch.delete(categoryToBeDeleted);
```

Figure 11: Lines 163-179 from the CategoryTable.js component file.

When a category is deleted, all the transactions within it should be deleted as well. For this purpose, Firestore's *writeBatch* function *(Firebase)* is used (Figure 11, Line 164). This function performs batched delete operations (Figure 11, Lines 175 and 179) to ensure data integrity, which is very important as it holds the client's vital financial data.

During the batched database operation processes mentioned previously, in the event that the client's system loses internet connection; experiences a failure; or there is an error in the database itself, the entire process is terminated – this ensures data integrity and hence reduces the chance of errors arising in the database.

Thus, through the *writeBatch* function, the CategoryTable component is able to fulfil an aspect of Criterion 6 – specifically that the client should be able to delete transaction categories. This enables the client to efficiently manage his transaction categories and, in extension, his transactions as well.

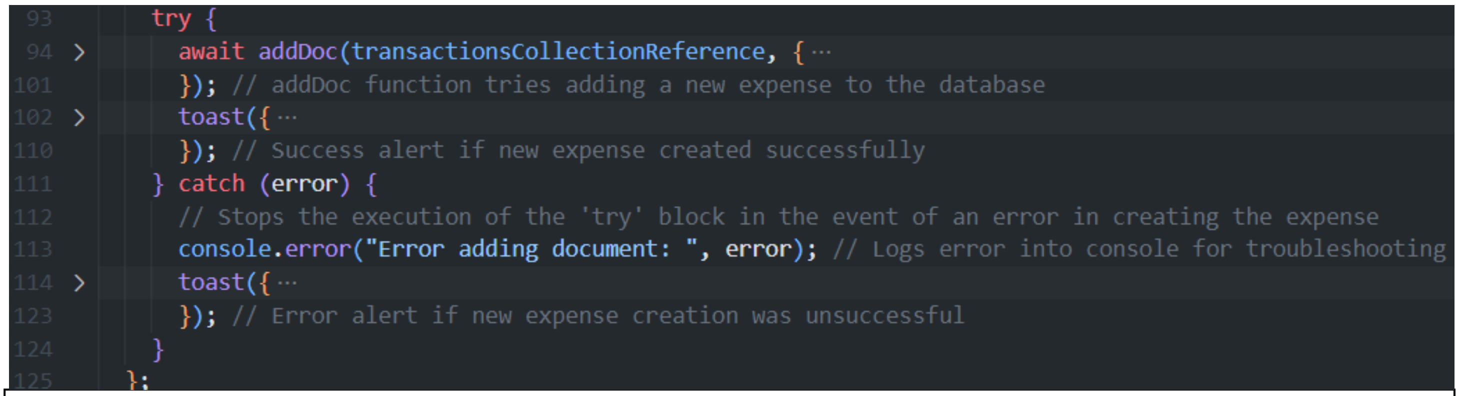


Delete category: *Category 1*?

Doing this will also delete all the transactions in the category.

Figure 12: Screen capture of the DeleteTransactionsModal component, which calls the handleDeleteCategory function (Figure 10) in CategoryTable when client clicks on 'Delete'.

2 Exception Handling using Asynchronous JavaScript



```
93       try {
94         await addDoc(transactionsCollectionReference, {…
101        }); // addDoc function tries adding a new expense to the database
102        toast({…
110        }); // Success alert if new expense created successfully
111      } catch (error) {
112        // Stops the execution of the 'try' block in the event of an error in creating the expense
113        console.error("Error adding document: ", error); // Logs error into console for troubleshooting
114        toast({…
123        }); // Error alert if new expense creation was unsuccessful
124      }
125    };
```

Figure 13: Lines 93-125 from the ExpenseForm.js component file. Some functions are only partially shown for better readability.

By using asynchronous JavaScript and try-catch blocks, the application is able to manage errors that may arise during database or authentication operations in a graceful manner. This approach is important for fulfilling Criterion 10 as it ensures the client is informed of any issues that may arise during database/authentication operations when he creates, edits or deletes transactions or when he signs-in using one of his Gmail accounts, hence enhancing the client's accessibility and experience even during events such as network disruptions, system failure, or database/authentication errors.

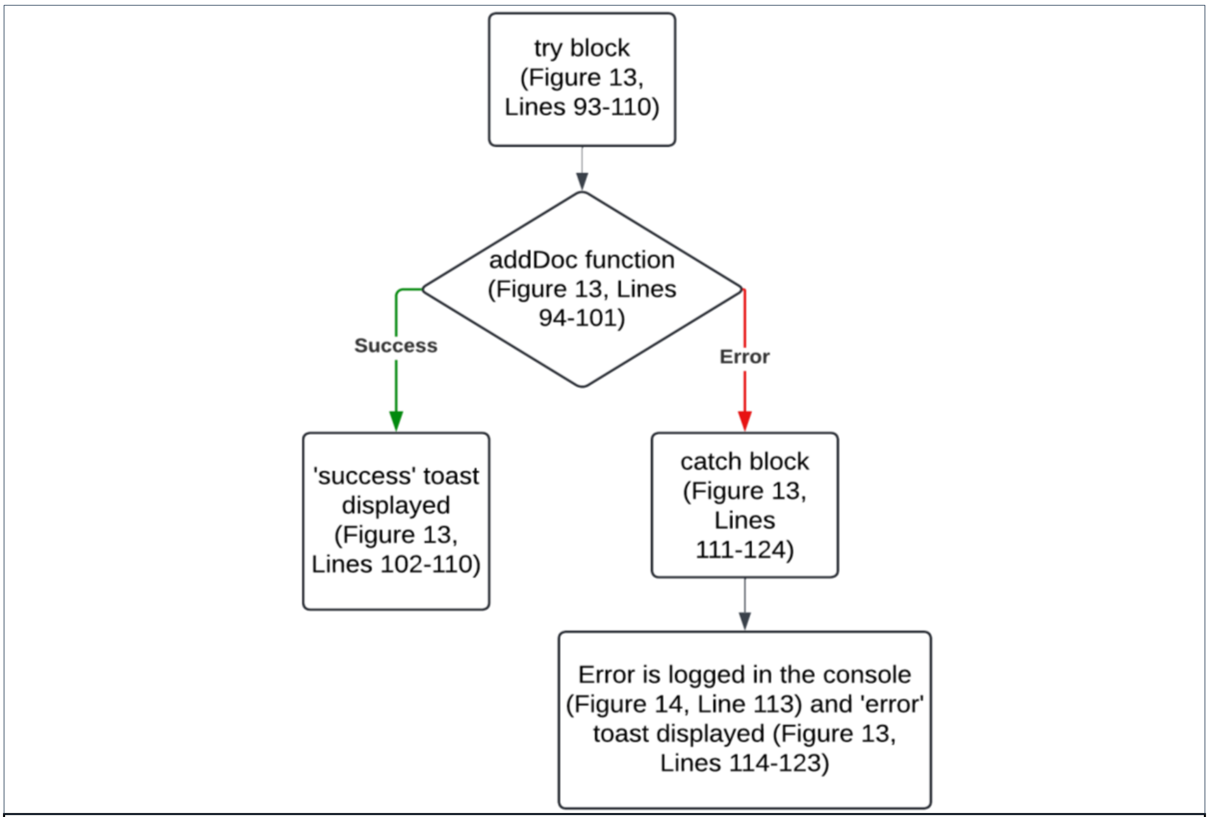
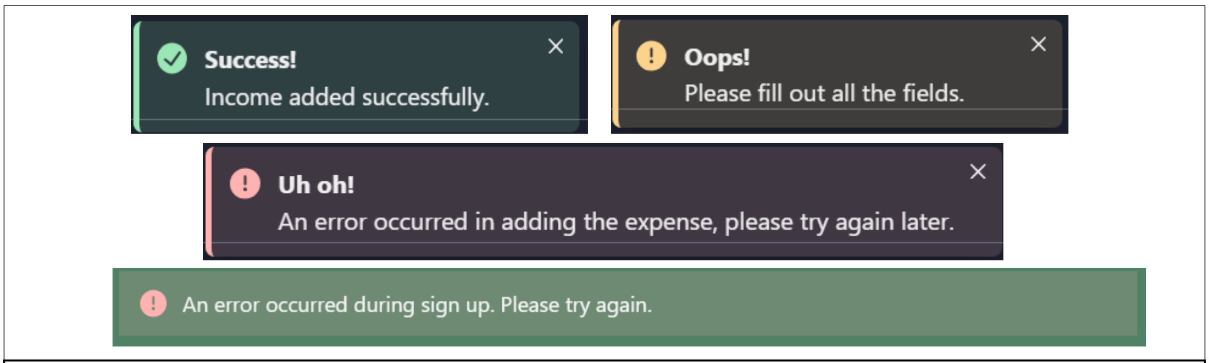

try block (Figure 13, Lines 93-110)
addDoc function (Figure 13, Lines 94-101)
Success
Error
'success' toast displayed (Figure 13, Lines 102-110)
catch block (Figure 13, Lines 111-124)
Error is logged in the console (Figure 14, Line 113) and 'error' toast displayed (Figure 13, Lines 114-123)

Figure 14: Process flowchart detailing the workings of the exception handling technique (based on the code in Figure 13).


Success!
Income added successfully.

Oops!
Please fill out all the fields.

Uh oh!
An error occurred in adding the expense, please try again later.

An error occurred during sign up. Please try again.

Figure 15: Screen captures of the ChakraUI (ChakraUI) "toast" components in the application: 'success' (top left), 'warning' (top right), and 'error' (middle, bottom) status alerts. They improve client experience as discussed above and fulfil Criterion 10.

3 Input Validation

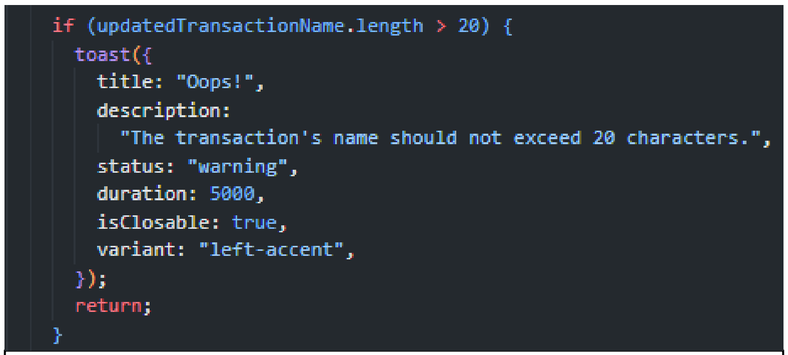

```
if (updatedTransactionName.length > 20) {
  toast({
    title: "Oops!",
    description:
      "The transaction's name should not exceed 20 characters.",
    status: "warning",
    duration: 5000,
    isClosable: true,
    variant: "left-accent",
  });
  return;
}
```

Figure 16: Lines 100-111 from the EditTransactionsModal.js component file

To maintain data integrity within Firestore, input validation techniques are implemented across all transaction and transaction category creating and editing related components. This technique is essential for Criteria 3, 5, and 6, as it ensures that only valid, normal data is entered into the database docs, preventing potential errors or inconsistencies within the database that could disrupt the client's important finance management processes. Criterion 10 is also fulfilled as client is alerted if a data validation error occurs (Figure 16, Lines 101-109).

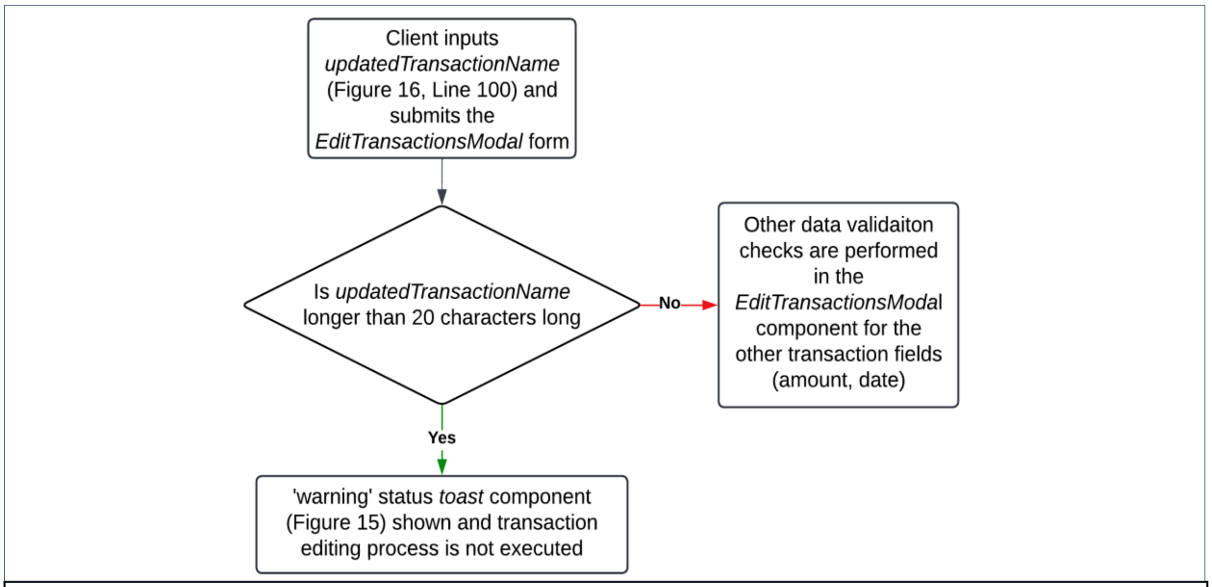
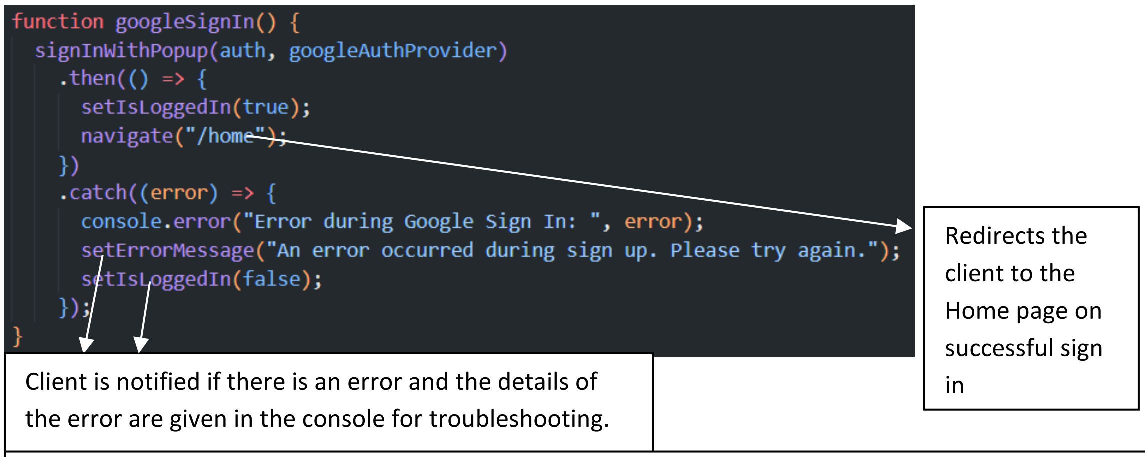

Client inputs *updatedTransactionName* (Figure 16, Line 100) and submits the *EditTransactionsModal* form
Is *updatedTransactionName* longer than 20 characters long
No
Other data validaiton checks are performed in the *EditTransactionsModal* component for the other transaction fields (amount, date)
Yes
'warning' status *toast* component (Figure 15) shown and transaction editing process is not executed

Figure 17: Process flowchart for the input validation of updated transaction names (based on the code in Figure 16).

4 Authentication and Verification



```
function googleSignIn() {
  signInWithPopup(auth, googleAuthProvider)
    .then(() => {
      setIsLoggedIn(true);
      navigate("/home");
    })
    .catch((error) => {
      console.error("Error during Google Sign In: ", error);
      setErrorMessage("An error occurred during sign up. Please try again.");
      setIsLoggedIn(false);
    });
}
```

Redirects the client to the Home page on successful sign in

Client is notified if there is an error and the details of the error are given in the console for troubleshooting.

Figure 18: Lines 26-37 from the SignIn.js page file.

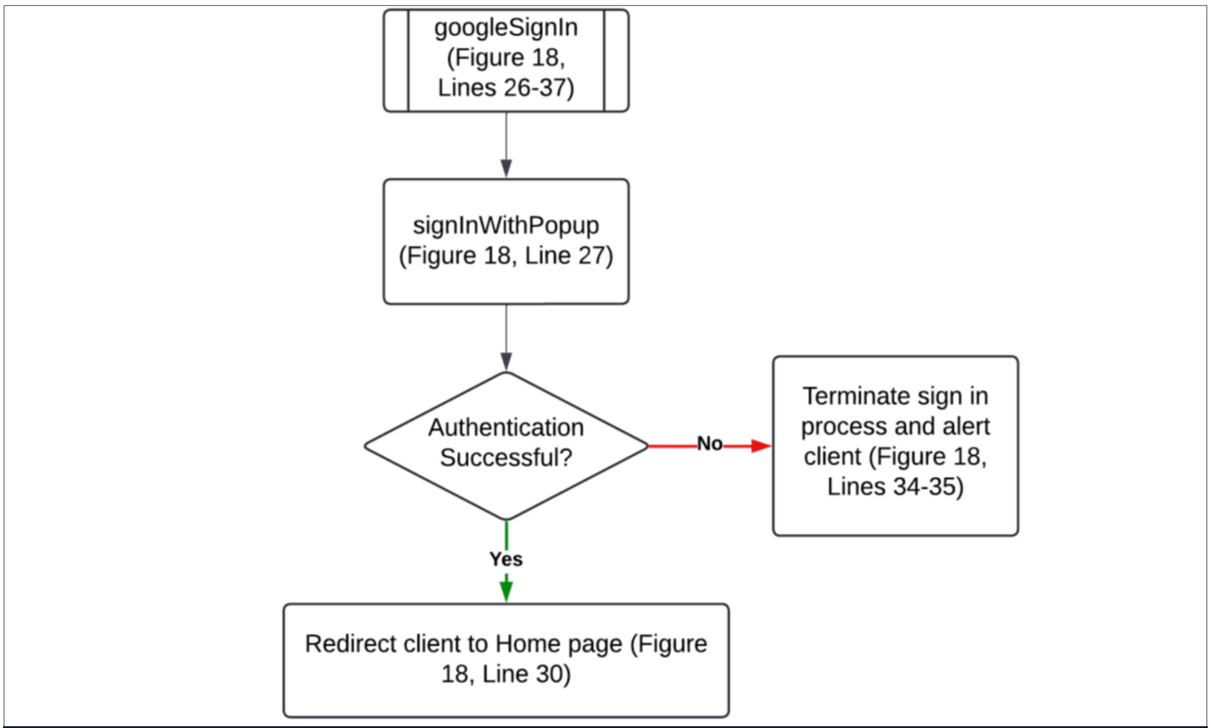
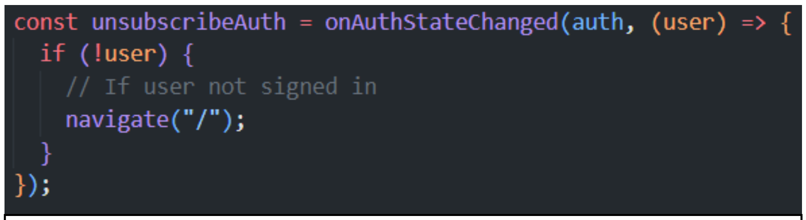

googleSignIn (Figure 18, Lines 26-37)
signInWithPopup (Figure 18, Line 27)
Authentication Successful?
No
Terminate sign in process and alert client (Figure 18, Lines 34-35)
Yes
Redirect client to Home page (Figure 18, Line 30)

Figure 19: Process flowchart detailing the workings of the googleSignIn function (based on the code in Figure 18).

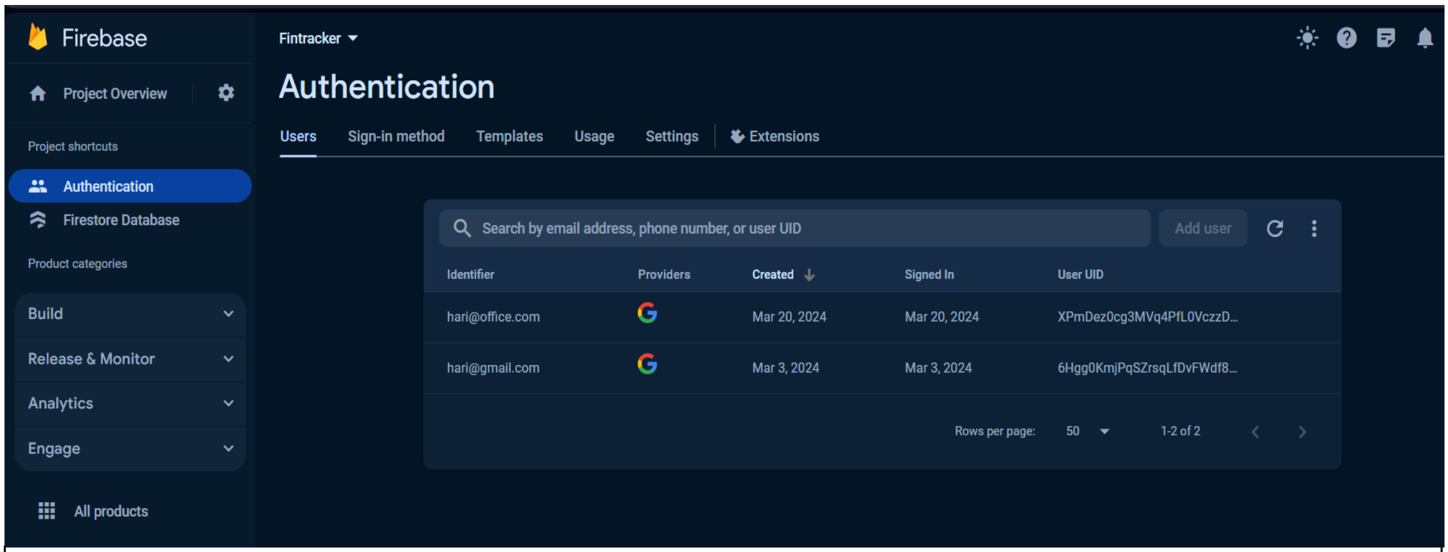
Since the *googleSignIn* function (Figures 18 and 19) displays the official authentication popup (Figure 19) which is displayed in both his Macbook and Windows systems, Mr. Hari can sign into his accounts with a single click on the screen (without having to input his password every time) – this feature is enabled only on his Macbook and Windows desktop as they are his frequently-used devices (however, this could be disabled on his Macbook as his children may gain unauthorized access to the application otherwise as Mr. Hari mentioned (Appendix A1)). This increases the conveniency with which the client can access the application while maintaining a high degree of security, thus further fulfilling Criterion 1, 2, and 9.



```
const unsubscribeAuth = onAuthStateChanged(auth, (user) => {
  if (!user) {
    // If user not signed in
    navigate("/");
  }
});
```

Figure 20: Lines 27-32 from the Reports.js page file.

Also, verification checks are done in each page through the Firebase *onAuthStateChanged* function *(Firebase)* (Figure 20, Lines 27-32) to enhance security by preventing unauthorized access to the application when the client is not signed into either one of his accounts – thus contributing to the fulfilment of Criterion 2.

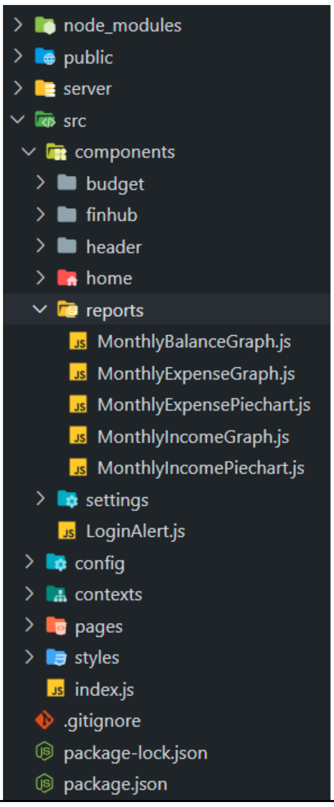


Authentication

Identifier	Providers	Created	Signed In	User UID
hari@office.com	G	Mar 20, 2024	Mar 20, 2024	XPmDez0cg3MVq4PfL0VczzD...
hari@gmail.com	G	Mar 3, 2024	Mar 3, 2024	6Hgg0KmjPqSZrsqLfDvFWdf8...

Figure 21: *Firebase stores a list of registered users. The client's privacy is secured as third-parties cannot access the Firebase console.*

5 Abstraction

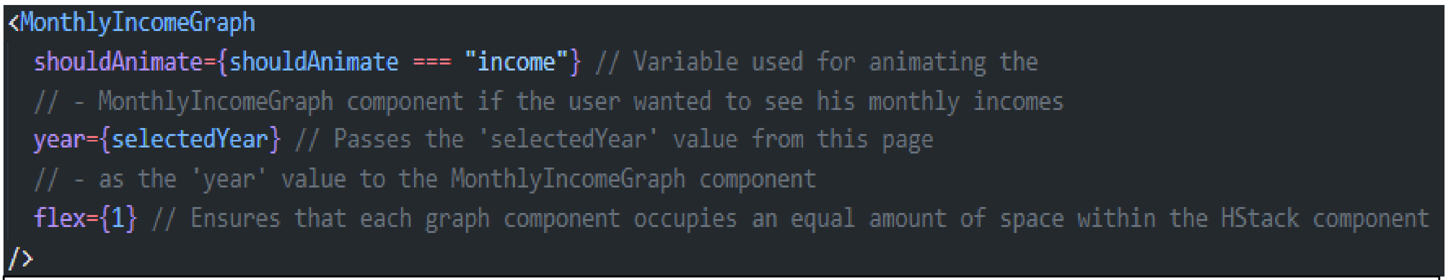


```
> node_modules
> public
> server
v src
  v components
    > budget
    > finhub
    > header
    > home
    v reports
        MonthlyBalanceGraph.js
        MonthlyExpenseGraph.js
        MonthlyExpensePiechart.js
        MonthlyIncomeGraph.js
        MonthlyIncomePiechart.js
    > settings
      LoginAlert.js
  > config
  > contexts
  > pages
  > styles
    index.js
  .gitignore
  package-lock.json
  package.json
```

Figure 22: Hierarchal overview of the modular/component-based structure of the product. The graph and chart component files used in the Reports page are shown.

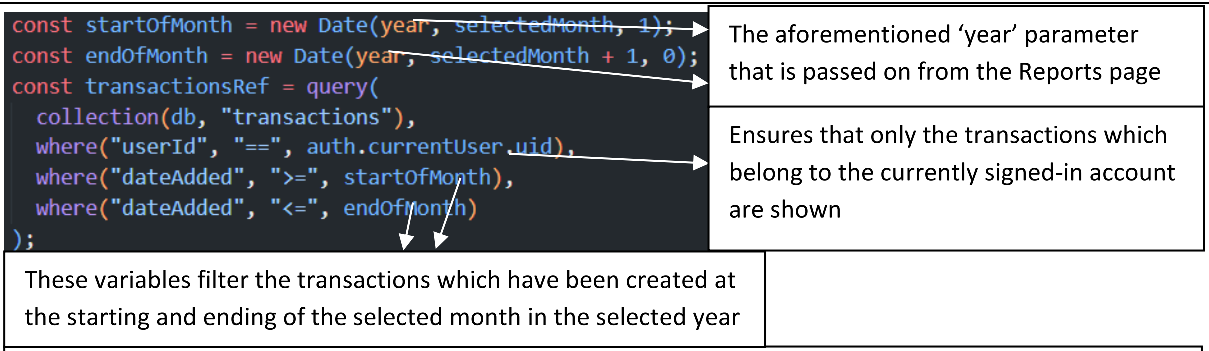
In the Reports page, each graph and piechart is rendered through a separate, dedicated component file instead of in the Reports page directly (Figure 22). The component takes in three parameters from the Reports page (Figure 23, Lines 89, 91, 93). One of them – the 'year' parameter – is set by the client in the Reports page and is passed on to the MonthlyIncomeGraph component, wherein it is used in the database query (Figure 24, Lines 64-65) to allow the client to see his income reports for specific months within the selected year.

This modular approach allows the client to easily switch the month for each graph independently without having to interact with the entire page, thus simplifying the client interface and enhancing user experience and effectively meeting Criterion 8. Moreover, the maintainability, usability, and scalability of the graph components are improved, hence further contributing to meeting Criterion 8 by enabling easy modifications and updates to the graphical representations of financial data.



```
<MonthlyIncomeGraph
  shouldAnimate={shouldAnimate === "income"} // Variable used for animating the
  // - MonthlyIncomeGraph component if the user wanted to see his monthly incomes
  year={selectedYear} // Passes the 'selectedYear' value from this page
  // - as the 'year' value to the MonthlyIncomeGraph component
  flex={1} // Ensures that each graph component occupies an equal amount of space within the HStack component
/>
```

Figure 23: Lines 88-94 from the Reports.js page file showing the use of abstraction.



```
const startOfMonth = new Date(year, selectedMonth, 1);
const endOfMonth = new Date(year, selectedMonth + 1, 0);
const transactionsRef = query(
  collection(db, "transactions"),
  where("userId", "==", auth.currentUser.uid),
  where("dateAdded", ">=", startOfMonth),
  where("dateAdded", "<=", endOfMonth)
);
```

The aforementioned 'year' parameter that is passed on from the Reports page

Ensures that only the transactions which belong to the currently signed-in account are shown

These variables filter the transactions which have been created at the starting and ending of the selected month in the selected year

Figure 24: Lines 64-71 from the MonthlyIncomeGraph.js component file.

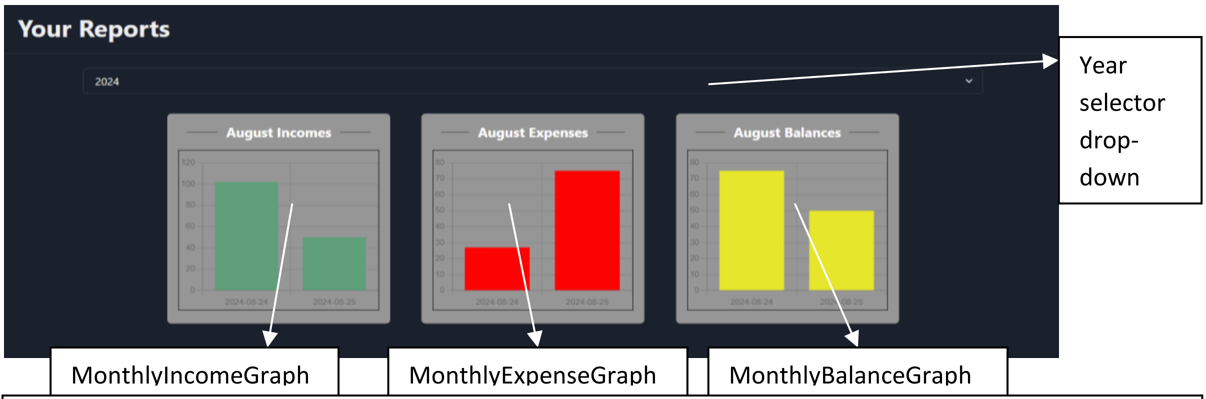
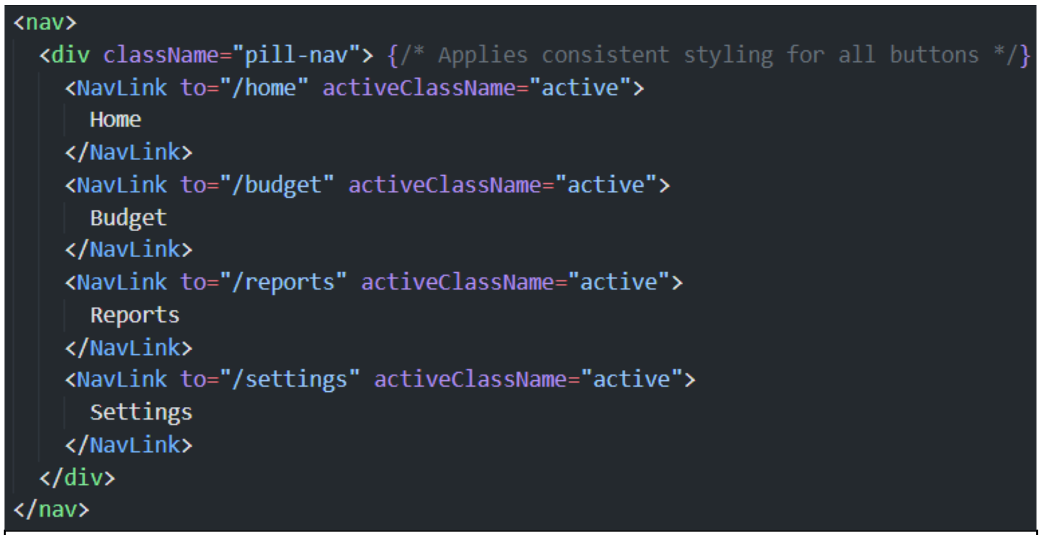


Figure 25: Partial screen capture of the Reports page. Abstraction was used in the aforementioned graph components in a similar manner as that of in MonthlyIncomeGraph as elaborated upon above.

6 Navigation



```
<nav>
  <div className="pill-nav"> {/* Applies consistent styling for all buttons */}
    <NavLink to="/home" activeClassName="active">
      Home
    </NavLink>
    <NavLink to="/budget" activeClassName="active">
      Budget
    </NavLink>
    <NavLink to="/reports" activeClassName="active">
      Reports
    </NavLink>
    <NavLink to="/settings" activeClassName="active">
      Settings
    </NavLink>
  </div>
</nav>
```

Figure 26: Lines 60-75 from the Header.js component file.

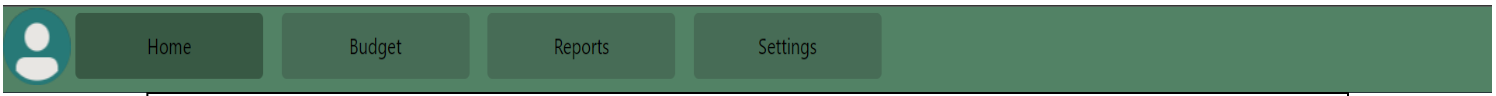
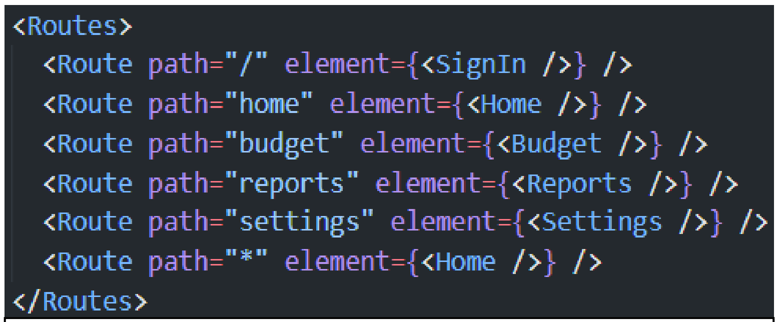


Figure 27: Screen capture of Header component in the Home page (which is why the 'Home' button is darkened, improving client experience and application navigability)



```
<Routes>
  <Route path="/" element={<SignIn />} />
  <Route path="home" element={<Home />} />
  <Route path="budget" element={<Budget />} />
  <Route path="reports" element={<Reports />} />
  <Route path="settings" element={<Settings />} />
  <Route path="*" element={<Home />} />
</Routes>
```

Figure 28: Lines 15-22 from the index.js main application file.

The use of the react-router-dom package's NavLink component *(Dorr and Strickland)* in the Header component (Figure 26, Lines 62-73) streamlines navigation across the application. This is because, when the client clicks on each button/link, the NavLink component redirects them to the requested page. For example, if the client clicks on "Budget", he is redirected to the "/budget" link (Figure 26, Line 65), which coresponds to the Budget page (Figure 28, Line 18) in application, thus redirecting him to that page.

This contributes to the overall fulfilment of Criteria 3, 4, 6, 7, and 8 as this technique ensures that the client can easily switch between performing his finance management-related tasks, such as adding transactions, filtering and finding specific transactions, viewing graph reports, or managing transaction categories.

Moreover, when the client clicks on the profile picture icon on the far-left in the Header (Figure 27), he can sign out of his current Gmail account and switch to his other account in the SignIn page, thus helping fulfil Criterion 1 (which states that he should be able to switch between his personal and professional Gmail accounts).

Word count: 1102

7 Bibliography

ChakraUI. *ChakraUI*. n.d. 10 October 2023. <https://v2.chakra-ui.com/>.

Dorr, Tim and Chance Strickland. *react-router-dom*. Vers. 6.22.3. n.d. 24 October 2023. <https://www.npmjs.com/package/react-router-dom>.

Firebase. *firebase*. Vers. 10.9.0. n.d. 5 October 2023. <https://www.npmjs.com/package/firebase>.

GeeksForGeeks. *Firestore and its advantages*. 19 February 2021. 6 October 2023. <https://www.geeksforgeeks.org/firestore-and-its-advantages/>.