Documentation for Financial Transactions Web Application

Jason N.

June 1, 2020

Contents

1 Disclaimer

This project is meant solely as a proof of concept to demonstrate how different databases might be used in this context. The project is NOT meant to be used in production. Several security flaws are present, including SQL injection, possible XSS, lack of authentication, etc.

2 Setup

This section is meant to serve as a general guide for setting up integrations used in this project. The detail in this guide is limited as the process will depend heavily on your choices, which I have attempted to outline for you. Many materials are referenced in this guide which contain far more detail, I would strongly suggest reading through these if they apply to your setup.

2.1 Google API

This is required for interacting with the Google Sheets database.

Go to https://console.developers.google.com/ and create a new project if you haven't already done that.

From the library panel, enable the Google Sheets API and the Google Drive API.

From the credentials panel, create an API key.

From the credentials panel, create an OAuth Client ID for a web application. Give it a name, which will appear when users are prompted to give the app permissions. Add the URIs that are expected to use the app. When testing this locally, it can be useful to add http://localhost:5000 or similar. These can always be changed at any moment from the developer console.

In the public/googleApiScript.js file of this repository, remember to change the client id and both instances of the api key to the appropriate values for your project.

2.2 Firebase

Firebase is used to host the web application and two of the databases, as well as storing images for all other databases to reference. All features are available through the same firebase project.

To get started, simply navigate to https://console.firebase.google.com and click "Add Project". Follow the instructions to set the name of the project and decide whether or not you want to make use of analytics.

Once a project has been created, follow the instructions at https://firebase.google.com/docs/web/setup to set up firebase with the front-end application. If you are using the files in this repository, the necessary SDKs are already included, though you'll need to change the firebase config to the appropriate values for your project.

To set up the real-time database, follow the instructions at https://firebase.google.com/docs/database/web/start to create a database and get your real-time database url. If you're using files from this repository, modify the firebase config to use this url instead of the given one.

To set up firestore, follow the instructions at https://firebase.google.com/docs/firestore/quickstart to create a database. If you're using files from this repository, modify the contents of the object passed to the

firebase.initializeApp() method to use appropriate values for your project. This method is called in the public/firebaseScript.js file.

To set up cloud functions, follow the instructions at https://firebase.google.com/docs/functions/get-started. If you're using files from this repository, the files already exist and just need to be deployed.

2.3 MySQL

There are several different implementations of MySQL available. MariaDB was used to create and test this project, which is a fork of MySQL.

If you decide to host the database yourself, you'll need to start the program, log in, and create a database. In this repository, it is named 'mydb', however, this can be changed if desired.

Remember to change the ip address, database name, and credentials in the index.js file of the firebase cloud functions folder.

If you wish to use this repository, the database can be imported using the dump.txt file:

```
mysql -u username -p database_name < dump.txt
```

Otherwise, once the database is created, enter the database using 'use database name;' to enter the database.

To create tables, you can use the following query:

```
CREATE TABLE ‘name‘ ( ‘colname1‘ datatype, ‘colname2‘ datatype, ‘colname3‘ datatype... )
```

Here is a useful website containing various MySQL commands: https://www.mysqltutorial.org/mysql-cheat-sheet.aspx/

3 HTML

3.1 Preamble and head

3.1.1 Preamble

Declares the document as HTML5.

```
1 <!DOCTYPE html>
```

3.1.2 meta charset

Specifies that characters in the file are encoded in UTF-8.

```
4 <meta charset = "UTF-8"/>
```

3.1.3 link rel=”stylesheet”

Imports the CSS file.

```
5 <link rel="stylesheet" type="text/css" href="./style.css"/>
```

3.1.4 Scripts

Imports the main Javascript file, responsible for the table and UI.

```
7 <script src="./script.js"></script>
```

Imports the Google API library.

```
8 <script src="https://apis.google.com/js/api.js"></script>
```

Imports other Javascript files, responsible for database management.

```
9  <script src="./googleApiScript.js"></script>
10 <script src="./mysqlScript.js"></script>
11 <script src="./localStorageScript.js"></script>
12 <script src="./imageFirestore.js"></script>
```

3.2 Inputs

Disables autocomplete which remembers past user input by default. `return false` specifies that no `POST` request should be made to the server.

```
20 <form onsubmit="return false" autocomplete="off">
```

3.2.1 Labels

Identifies the purpose of the field to the user, allows the user to select the field by clicking the label. This element is also used by accessibility tools to identify the field.

```
22 <label for="date">Date:</label><br/>
```

3.2.2 Date

The `date` input type is supported by most modern browsers and provides an intuitive UI for selecting dates. It also includes methods for converting or verifying the `Date` object.

```
13 <input id="date" name="date" type="date" placeholder="yyyy-mm-dd"/>
```

3.2.3 Text

The `text` input type allows the user to input a string. For numbers, this string has to be parsed in Javascript.

```
28 <input id="account" name="account" list="accountsList" type="text"
   ↪ placeholder="Account Number"/>
```

3.2.4 List

Lists are created using the `select` element, containing `option` elements. Each option has a `value` which is used in Javascript, and `innerText` which is seen by the user.

```
34 <label for="type">Transaction Type:</label><br/>
35 <select id="type" name="type">
36     <option value=""></option>
37     <option value="BUY">BUY</option>
38     <option value="SELL">SELL</option>
39     <option value="!DIVIDEND">DIVIDEND</option>
40     <option value="!INTEREST">INTEREST</option>
41     <option value="!WITHDRAW">WITHDRAW</option>
42     <option value="!DEPOSIT">DEPOSIT</option>
43 </select>
```

3.2.5 File

Files are uploaded using the `file` input type. The `multiple` attribute allows the user to upload multiple files, which are interpreted as an array of files in Javascript.

```
64     <label id="fileUploadLabel" for="fileUpload">Upload file</label>
65     <input id="fileUpload" name="fileUpload" type="file" onchange="
          ↪ fileUploadChanged();" multiple/>
```

3.2.6 Buttons

Buttons with the `submit` type can be used to check that all required sections are complete and highlight them in red. These buttons can also be used to send a `POST` request to a server if desired. The `onclick` attribute specified the function and parameters that should be executed when pressed.

```
70 <button id="add" type="submit" onclick="addTransactionButton();">Add
   ↪ Transaction</button>
```

3.3 Filters

Filter HTML elements are handled exactly the same as their counterparts in the input section. Some fields have two elements to handle a lower and upper bound, but these are handled solely in Javascript.

3.3.1 Tooltip

The `span` element is a generic container. The `title` attribute will display its value as a tool tip when the element is hovered.

```
118 <span title="Enter search terms here. Separate search terms with && or ||
    ↪ for AND and OR statements, respectively. Exclusive filters are
    ↪ marked by a leading !. Use || to filter by multiple securities (e.g.
    ↪  SPY || TLT) and && to exclude multiple securities (e.g. !SPY && !
    ↪ TLT).">?</span><br/>
```

3.3.2 Checkbox

The input type `checkbox` provides a toggleable input field which can be evaluated as `true` or `false` with Javascript.

```
153 <label for="filterNa">Filter N/A:</label>
154 <input id="filterNA" name="filterNA" type="checkbox"/>
```

3.4 Options

The options section uses buttons, text inputs, a file input, and drop down menus, which are decribed in the inputs section. The special handling of these elements is done in Javascript.

3.5 Table

3.6 frozenColumns

Cells in columns that are meant to be always visible are marked with a `frozenColumnx` class, where `x` is the column number. CSS is used to keep the column in place when scrolling.

```
<th class="frozenColumn1">
    <section>
        Transaction ID
    </section>
```

3.7 sort buttons

Sorting is done using buttons with an `onclick` attribute that calls a function `sortTable()`. The parameters passed are the column index and a boolean value indicating whether the column should be sorted in ascending or descending order.

```
<section class="sort">
    <button type="button" onclick="sortTable(0, true)">^</button>
    <button type="button" onclick="sortTable(0, false)">v</button>
</section>
```

3.7.1 tbody

The main table body is initially empty. Rows are managed by Javascript and it is marked with a unique id for this purpose.

```
<tbody id="tableBody">
</tbody>
```

3.8 Firebase scripts

These scripts are taken directly from the firebase documentation. They are required for firebase and its components to function. The `firebase-app.js` script is the main script and is required for all firebase features. The next three scripts are required for collecting analytics data, the realtime database, and firestore, respectively.

The configuration contains API keys and project information required to identify the app. The key is not secret, though it is unique to the project. As it is easily obtained by users of the app, it is strongly recommended to whitelist your domain in the project settings.

Unlike the other scripts, the firebase script is declared at the bottom, as it requires that the SDKs have loaded first.

```
<!-- The core Firebase JS SDK is always required and must be listed first
    ↪ -->
<script src="https://www.gstatic.com/firebasejs/7.14.2/firebase-app.js"></
    ↪ script>

<script src="https://www.gstatic.com/firebasejs/7.14.2/firebase-analytics.
    ↪ js"></script>
<script src="https://www.gstatic.com/firebasejs/7.14.2/firebase-database.
    ↪ js"></script>
<script src="https://www.gstatic.com/firebasejs/7.14.3/firebase-firestore.
    ↪ js"></script>
```

```
329
330 <script>
331 // Your web app's Firebase configuration
332 var firebaseConfig = {
333     apiKey: "AIzaSyAmZLFZHDAB9evhvNunxOe5GxXRd_OizmU",
334     authDomain: "financial-transactions-6f065.firebaseapp.com",
335     databaseURL: "https://financial-transactions-6f065.firebaseio.com",
336     projectId: "financial-transactions-6f065",
337     storageBucket: "financial-transactions-6f065.appspot.com",
338     messagingSenderId: "82206982479",
339     appId: "1:82206982479:web:8937bbd1bd4fb6022b053a",
340     measurementId: "G-0564DT8RNQ"
341 };
342 // Initialize Firebase
343 firebase.initializeApp(firebaseConfig);
344 firebase.analytics();
345
346 var database = firebase.database();
347 var firestore = firebase.firestore();
348 </script>
349
350 <script src="./firebaseScript.js"></script>
```

4 Main Javascript

This file handles the UI and general functions required to bridge the front end with the databases.

4.1 formattedStringToNumber()

Removes leading dollar sign if present. Removes all commas. Converts string to a number datatype.

```
1  function formattedStringToNumber(numberAsString) {
2      var number;
3
4      if(numberAsString[0] == '$') {
5          numberAsString = numberAsString.substr(1);
6      }
7
8      number = Number(numberAsString.replace(/,/g, ''));
9
10     return number;
11 }
```

4.2 numberToFormattedString()

Converts number to string datatype. Inserts a comma between every consecutive group of 3 characters.

```
13 function numberToFormattedString(number) {
14     var numberAsString;
15
16     numberAsString = String(number).replace(/\B(?=(\d{3})+(?!\d))/g, ",");
17
18     return numberAsString;
19 }
```

4.3 getData()

Gets values from input fields and performs minor formatting changes. Calls the `validate()` function to have the data verified. If the data is valid, more formatting changes are performed, including adding dollar signs and converting the date to a string. The function returns an array of the data if valid, `false` otherwise.

```
21 function getData() {
22     var date = document.getElementById("date");
23     var account = document.getElementById("account").value;
24     var type = document.getElementById("type").value;
25     var security = document.getElementById("security").value;
26     var amount = document.getElementById("amount").value;
27     var dAmount = document.getElementById("dAmount").value;
28
29     security = security.toUpperCase();
30
31     amount = formattedStringToNumber(amount);
```

```
32
33      dAmount = formattedStringToNumber(dAmount);
34
35      if(validate(date, account, type, security, amount, dAmount)) {
36          var costBasis = '$' + numberToFormattedString(calculateCostBasis(
                ↪ amount, dAmount));
37          date = date.value;
38
39          amount = numberToFormattedString(amount);
40          dAmount = '$' + numberToFormattedString(dAmount.toFixed(2));
41
42          return [ date, account, type, security, amount, dAmount, costBasis
                ↪ ];
43      }
44      else return false;
45  }
```

4.4 validate()

Calls functions to validate all input fields. If any return `false`, the `validate()` function returns `false`. If none of the checks fail, the function returns `true`.

```
47  function validate(date, account, type, security, amount, dAmount) {
48      if(!validateDate(date)) return false;
49      if(!validateAccount(account)) return false;
50      if(!validateType(type)) return false;
51      if(!validateSecurity(security)) return false;
52      if(!validateAmount(amount)) return false;
53      if(!validateDAmount(dAmount)) return false;
54
55      return true;
56  }
```

4.4.1 Check empty

Checks if the input field is an empty string. If so, alerts the user with an error message and returns `false`. Otherwise, returns `true`.

```
80  function validateAccount(account) {
81      if(account == '') {
82          alert('Error: Missing Account Number');
83          return false;
84      }
85
86      return true;
87  }
```

4.4.2 Check NaN

Uses the built-in `isNaN()` function to check that a number is valid.

```
107 function validateAmount(amount) {
108     if(amount == '') {
109         alert('Error: Missing Amount');
110         return false;
111     }
112
113     if(isNaN(amount)) {
114         alert('Error: Invalid Amount');
115         return false;
116     }
117
118     return true;
119 }
```

4.4.3 Check date

Gets the current date and stores it in the variable `realDate`. Checks the validity of the date input using the built-in `date.checkValidity()`. Compares the date input to the current date to ensure that the date input is not in the future.

```
58 function validateDate(date) {
59     realDate = new Date();
60     inputDate = date.valueAsNumber;
61
62     if(date.value == '') {
63         alert('Error: Missing date');
64         return false;
65     }
66
67     if(!date.checkValidity()) {
68         alert('Error: Invalid date');
69         return false;
70     }
71
72     if(realDate.valueOf() < inputDate) {
73         alert('Error: Date is in the future');
74         return false;
75     }
76
77     return true;
78 }
```

4.5 generateId()

4.6 calculateCostBasis()

4.7 addTransaction()

4.8 fileIdGenerator()

4.9 addTransactionButton()

4.10 addTransactionWithFileName()

4.11 deleteRow()

4.12 editRow()

4.13 saveChanges()

4.14 discardChanges()

4.15 sortTable()

4.16 resetDate()

4.17 validateFilters()

4.17.1 always true

4.17.2 date range

4.17.3 amount range

4.17.4 generic ranges

4.18 stringFilter()

4.19 applyFilter()

4.20 clearFilter()

4.21 unfilterAll()

4.22 toggleID()

4.23 readFile()

4.24 saveFile()

5 firebaseScript.js

5.1 clearFirebase()

5.2 writeToFirebase()

5.3 readFromFirebase()

5.4 clearFirestore()

5.5 writeToFirestore()

5.6 readFromFirestore()

6 googleApiScript.js

6.1 Global Variables

6.2 loadSheetData()

6.3 getNewSheetData()

6.4 populateSheetSelector()

6.5 getNewTabData()

6.6 populateTabSelector()

6.7 getAllUserSheets()

6.8 getTabsOfSheet()

6.9 authenticate()

6.10 loadClientSheets()

6.11 loadClient()

6.12 readGoogleSheetDB()

6.13 readGoogleTypes()

6.14 writeGoogleSheetDB()

6.15 setGoogleRows()

6.16 clearGoogleRow()

6.17 writeGoogleDB()

6.18 gapi.load()

7 imageFirestore.js

7.1 writeImagesToFirestore()

7.2 readImagesFromFirestore()

7.3 getFileNamesIds()

7.4 parseFileNamesIds()

7.5 clearIndexedDb()

8 localStorageScript.js

8.1 Global Variables

8.2 initDb()

8.3 fileUploadChanged()

8.4 uploadFile()

8.5 addFile()

8.6 updateExistingFileName()

8.7 removeFileFromTable()

8.8 deleteFileFromIndexedDB()

8.9 removeFileUpload()

8.10 downloadFile()

8.11 window.onbeforeunload = function()

9 mysqlScript.js

9.1 writeToMySQL()

9.2 readFromMySQL()

10 CSS

10.1 Vertical Scrolling Table

10.2 Horizontal Scrolling on Overflow

10.3 Miscellaneous

10.3.1 Sort buttons

10.3.2 Editing highlight

10.3.3 Table borders