

Documentation for Financial Transactions Web Application
Jason N.
June 1, 2020

Contents

1	Disclaimer	1
2	Setup	1
2.1	Google API	1
2.2	Firebase	1
2.3	MySQL	2
3	HTML	3
3.1	Preamble and head	3
3.1.1	Preamble	3
3.1.2	meta charset	3
3.1.3	link rel="stylesheet"	3
3.1.4	Scripts	3
3.2	Inputs	3
3.2.1	Labels	4
3.2.2	Date	4
3.2.3	Text	4
3.2.4	List	4
3.2.5	File	4
3.2.6	Buttons	5
3.3	Filters	5
3.3.1	Tooltip	5
3.3.2	Checkbox	5
3.4	Options	5
3.5	Table	5
3.6	frozenColumns	5
3.7	sort buttons	6
3.7.1	tbody	6
3.8	Firebase scripts	6
4	Main Javascript	8
4.1	formattedStringToNumber()	8
4.2	numberToFormattedString()	8
4.3	getData()	8
4.4	validate()	9
4.4.1	Check empty	9
4.4.2	Check NaN	10
4.4.3	Check date	10
4.5	generateId()	10
4.6	calculateCostBasis()	11
4.7	addTransaction()	11
4.8	fileIdGenerator()	13
4.9	addTransactionButton()	13
4.10	addTransactionWithFileName()	13
4.11	deleteRow()	14
4.12	editRow()	14
4.13	saveChanges()	15
4.14	discardChanges()	16
4.15	sortTable()	17
4.16	resetDate()	17
4.17	clearInput()	18
4.18	validateFilters()	18
4.18.1	always true	18

4.18.2	date range	19
4.18.3	amount range	19
4.19	stringFilter()	20
4.20	applyFilter()	20
4.21	clearFilter()	22
4.22	unfilterAll()	22
4.23	toggleID()	22
4.24	readFile()	23
4.25	saveFile()	24
4.26	applyTypes()	24
4.27	editTypes()	24
4.28	setTransactionTypes()	25
4.29	toggleSection()	25
4.30	loadDataLists()	25
4.31	readCurrentTypes()	26
4.32	tableToArrays()	27
4.33	arraysToTable()	27
4.34	window.onload = function()	28
5	firebaseScript.js	29
5.1	clearFirebase()	29
5.2	writeToFirebase()	29
5.3	readFromFirebase()	29
5.4	clearFirestore()	30
5.5	writeToFirestore()	31
5.6	readFromFirestore()	32
6	googleApiScript.js	34
6.1	Global Variables	34
6.2	loadSheetData()	34
6.3	getNewSheetData()	34
6.4	populateSheetSelector()	35
6.5	getNewTabData()	35
6.6	populateTabSelector()	35
6.7	getAllUserSheets()	36
6.8	getTabsOfSheet()	36
6.9	authenticate()	36
6.10	loadClientSheets()	37
6.11	loadClient()	37
6.12	readGoogleSheetDB()	37
6.13	readGoogleTypes()	38
6.14	writeGoogleSheetDB()	38
6.15	setGoogleRows()	38
6.16	clearGoogleRow()	39
6.17	writeGoogleDB()	39
6.18	gapi.load()	40
7	imageFirestore.js	41
7.1	writeImagesToFirestore()	41
7.2	readImagesFromFirestore()	41
7.3	getFileNamesIds()	42
7.4	parseFileNamesIds()	42
7.5	clearIndexedDb()	43

8	localStorageScript.js	44
8.1	Global Variables	44
8.2	initDb()	44
8.3	fileUploadChanged()	45
8.4	uploadFile()	45
8.5	addFile()	46
8.6	updateExistingFileName()	47
8.7	removeFileFromTable()	48
8.8	deleteFileFromIndexedDB()	48
8.9	removeFileUpload()	49
8.10	downloadFile()	49
8.11	window.onbeforeunload = function()	50
9	mysqlScript.js	51
9.1	writeToMySQL()	51
9.2	readFromMySQL()	51
10	NodeJs	53
10.1	Dependencies	53
10.2	mysql.createConnection()	53
10.3	con.connect()	53
10.4	readDataFromMySQL()	53
10.5	readTypesFromMySQL()	53
10.6	writeToMySQL()	54
10.7	api.post()	54
10.8	api.get()	54
10.9	exports.api	54
11	CSS	55
11.1	Vertical Scrolling Table	55
11.2	Horizontal Scrolling on Overflow	55
11.3	Miscellaneous	55
11.3.1	Sort buttons	55
11.3.2	Editing highlight	55
11.3.3	Table borders	55

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 described 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.

```

233 <th class="frozenColumn1">
234   <section>
235     Transaction ID
236   </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.

```

237 <section class="sort">
238   <button type="button" onclick="sortTable(0, true)">^</button>
239   <button type="button" onclick="sortTable(0, false)">v</button>
240 </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.

```

317 <tbody id="tableBody">
318 </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.

```

323 <!-- The core Firebase JS SDK is always required and must be listed first
    ↪ -->
324 <script src="https://www.gstatic.com/firebasejs/7.14.2/firebase-app.js"></
    ↪ script>
325
326 <script src="https://www.gstatic.com/firebasejs/7.14.2/firebase-analytics.
    ↪ js"></script>
327 <script src="https://www.gstatic.com/firebasejs/7.14.2/firebase-database.
    ↪ js"></script>
328 <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_0izmU",
334   authDomain: "financial-transactions-6f065.firebaseio.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 = '$' + numberToFormattedString(calculateCostBasis(
37             ↪ amount, dAmount));
38         date = date.value;
39
40         amount = numberToFormattedString(amount);
41         dAmount = '$' + numberToFormattedString(dAmount.toFixed(2));
42
43         return [ date, account, type, security, amount, dAmount, costBasis
44             ↪ ];
45     }
46     else return false;
47 }

```

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()

Generates an ID of length `idLength` by selecting a random character from the character set using a loop. Checks this ID against all other IDs in the table, if none match, the function returns the ID. If any match, the ID is not unique and the function attempts to generate another until it reaches a unique ID.

```

135 function generateId() {
136     var id = '';
137     var idLength = 6;
138
139     var characters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789';
140     var charactersLength = characters.length;
141
142     var unique = false;
143
144     while(!unique) {
145         for(var i = 0; i < idLength; i++) {
146             id += characters.charAt(Math.floor(Math.random() *
147                 ↪ charactersLength));
148         }
149
150         unique = true;
151         for(var i = 0; i < document.getElementsByClassName('idCell').
152             ↪ length; i++) {
153             if(document.getElementsByClassName('idCell')[i].innerText ==
154                 ↪ id) {
155                 unique = false;
156                 break;
157             }
158         }
159     }
160     return id;
161 }

```

4.6 calculateCostBasis()

Divides the dollar amount by the amount and sets precision to 2 decimal places.

```

160 function calculateCostBasis(amount, dAmount) {
161     costBasis = (dAmount / amount).toFixed(2);
162     return costBasis;
163 }

```

4.7 addTransaction()

Takes an array as the argument, meant to contain all data necessary to create a row. Constructs remaining cells in the row, such as the actions column, and formats the files column. Adds the new row to the table and adds cells, modifying classes where necessary.

```

165 function addTransaction(data) {
166     var staging = data;
167     var tableBody = document.getElementById('tableBody');
168     var newRow = tableBody.insertRow(0);
169     newRow.classList += "bodyRow";
170 }

```

```

171     var actionsContent = "<button type='button' onclick='editRow(this)''>
    ↪ Edit</button><button type='button' onclick='deleteRow(this)''>
    ↪ Delete</button>";
172     var fileContent = '<table><tbody>';
173     if(data.length > 8) {
174         for(let i = 0; i < data[8].length; i++) {
175             fileContent += "<tr><td><a onclick='downloadFile('" + data[8][
    ↪ i][0] + "';' href='javascript:void(0);''>" + data[8][i
    ↪ ][1] + "</a></td>";
176             fileContent += "<td><button type='button' onclick='
    ↪ removeFileFromTable('" + data[8][i][0] + "', this);''>-</
    ↪ button></td></tr>";
177         }
178     }
179     fileContent += '</tbody></table><input type="file" onchange="addFile(
    ↪ this, 0);" multiple/>';
180     staging[8] = fileContent;
181     staging[9] = actionsContent;
182
183     var calculateCostBasis = true;
184     if(data[3][0] == '!') {
185         calculateCostBasis = false;
186         data[3] = data[3].substr(1);
187     }
188
189     for(var i = 0; i < 10; i++) {
190         var newCell = newRow.insertCell(i);
191         var idShowing = (document.getElementById('toggleId').innerText ==
    ↪ "Hide Transaction ID");
192
193         newCell.innerHTML = data[i];
194
195         if(i == 0) {
196             if(idShowing)
197                 newCell.classList = "idCell frozenColumn1";
198             else {
199                 newCell.classList = "idCell";
200                 newCell.setAttribute("hidden", true);
201             }
202         }
203         else if(i == 1) {
204             if(idShowing)
205                 newCell.classList = "frozenColumn2";
206             else
207                 newCell.classList = "frozenColumn1";
208         }
209         /*
210         else if(i == 2) {
211             if(idShowing)
212                 newCell.classList = "frozenColumn3";
213             else
214                 newCell.classList = "frozenColumn2";
215         }
216         */

```



```
217
218         if(i == 7 && !calculateCostBasis) {
219             newCell.innerHTML = "N/A";
220         }
221     }
222 }
```

4.8 fileIdGenerator()

Creates and arbitrary, random file ID, large enough that it is extremely unlikely to generate two identical IDs. This is done as it is less feasible to check existing databases for matching IDs.

[illegible]

4.9 addTransactionButton()

Gets data by calling the `getData()` function. Adds the transaction ID to the `data` array. Creates a new empty array for storing file data, calls the `uploadFile()` function to continue the process.

```
228 function addTransactionButton() {
229     var data = getData();
230     if(data) {
231         var id = generateId();
232         data.unshift(id);
233
234         fileList = new Array()
235         uploadFile(data, addTransactionWithFileName, fileList, 0);
236         clearInput(false);
237     }
238 }
```

4.10 addTransactionWithFileName()

Takes the row data as an argument and calls `addTransaction()` to finalise the process. Logs the data in the Javascript console. Calls the `loadDataLists()` function to update the list of securities and accounts lists.

```
240 function addTransactionWithFileName(data) {
241     addTransaction(data);
242     console.log(data);
243     loadDataLists();
244 }
```

4.11 deleteRow()

Gets the row of the delete button. Removes files from the local database which were referenced from this row (using the unique file ID). Removes the row. Resets the buttons in the input section if necessary (i.e. if the deleted row was being edited, the editing actions must be hidden and the add transaction button must be restored).

```
246 function deleteRow(button) {
247     var row = button.parentElement.parentElement;
248
249     var fileRows = row.getElementsByTagName('td')[8].getElementsByTagName
    ↪ ('table')[0].getElementsByTagName('tr');
250     for(let i = 0; i < fileRows.length; i++) {
251         let fileId = fileRows[i].getElementsByTagName('a')[0].getAttribute
    ↪ ('onclick').split(' ')[1];
252         deleteFileFromIndexedDB(fileId);
253     }
254
255     document.getElementById("tableBody").removeChild(row);
256
257     if(document.getElementsByClassName('editing').length == 0) {
258         document.getElementById('add').removeAttribute('hidden');
259         document.getElementById('save').setAttribute('hidden', true);
260         document.getElementById('discard').setAttribute('hidden', true);
261
262         document.getElementById('add').setAttribute('type', 'submit');
263         document.getElementById('save').setAttribute('type', 'button');
264     }
265     loadDataLists();
266 }
```

4.12 editRow()

The function checks if there is already a row with the `editing` class. If so, it removes this class from the old row.

The function then sets the current row to have the `editing` class. All the input fields are set to use values from the row.

In the case of type, the function first tries to apply the display value of the type. If this fails, the function tries to add an exclamation mark to the front to account for the indication of types without cost basis.

The add transaction button is hidden and set to a regular button so that the enter key no longer activates it. The save and discard buttons are unhidden, and save is set to the submit type button.

The file upload is cleared and the content of the file upload label is set depending on the number of files from the row.

```
268 function editRow(button) {
269     if(document.getElementsByClassName('editing').length > 0)
270         document.getElementsByClassName('editing')[0].classList = "bodyRow
    ↪ ";
271 }
```

```

272     var row = button.parentElement.parentElement;
273     var rowContent = row.getElementsByTagName('td');
274     row.classList = "bodyRow editing";
275
276     document.getElementById('date').value = rowContent[1].innerText;
277     document.getElementById('account').value = rowContent[2].innerText;
278
279     document.getElementById('type').value = rowContent[3].innerText;
280     if(document.getElementById('type').value == '') document.
        ↪ getElementById('type').value = '!' + rowContent[3].innerText;
281
282     document.getElementById('security').value = rowContent[4].innerText;
283     document.getElementById('amount').value = rowContent[5].innerText;
284     document.getElementById('dAmount').value = rowContent[6].innerText;
285
286     document.getElementById('add').setAttribute('hidden', true);
287     document.getElementById('save').removeAttribute('hidden');
288     document.getElementById('discard').removeAttribute('hidden');
289
290     document.getElementById('add').setAttribute('type', 'button');
291     document.getElementById('save').setAttribute('type', 'submit');
292
293     removeFileUpload();
294     uploadLabel = document.getElementById('fileUploadLabel');
295     if(rowContent[8].getElementsByTagName('tr').length > 0) {
296         uploadLabel.innerHTML = String(rowContent[8].getElementsByTagName
        ↪ ('tr').length) + " file(s)";
297     }
298     fileEditted = false;
299 }

```

4.13 saveChanges()

The function attempts to get data from the input section by calling `getData()`, which also validates this data.

The row to modify is determined by checking for the `editing` class. An array is created from the cells in this row.

If the first character of the type is and exclamation mark, this character is removed before being inserted into the row, and the cost basis is set to N/A.

Each element in the data array is moved into the appropriate cell in the row.

The add transaction button is restored, and the other buttons are hidden.

If the file upload was changed (i.e. a file was uploaded or the element was cleared), the `uploadFile()` function is called to get the new files if they exist. If no changes were made, the file input is cleared.

The date in the input is reset and all other fields are cleared. Lastly, the function loads existing accounts and securities for the autofill feature.

```

301 function saveChanges() {

```

```

302     data = getData();
303     if(data) {
304         rowToEdit = document.getElementsByClassName('editing')[0];
305         cellsToEdit = rowToEdit.getElementsByTagName('td');
306
307         if(data[2][0] == '!') {
308             data[2] = data[2].substr(1);
309             data[6] = "N/A";
310         }
311
312         for(var i = 0; i < data.length; i++) {
313             cellsToEdit[i + 1].innerHTML = data[i];
314         }
315         rowToEdit.classList = "bodyRow";
316
317         document.getElementById('add').removeAttribute('hidden');
318         document.getElementById('save').setAttribute('hidden', true);
319         document.getElementById('discard').setAttribute('hidden', true);
320
321         document.getElementById('add').setAttribute('type', 'submit');
322         document.getElementById('save').setAttribute('type', 'button');
323
324         if(fileEditted) {
325             uploadFile([cellsToEdit[8]], updateExistingFileName, new Array
326                 ↪      (), 0);
327         }
328         else {
329             removeFileUpload();
330         }
331
332         resetDate();
333         clearInput(true);
334         loadDataLists();
335     }
336 }

```

4.14 discardChanges()

The `editing` class is removed from the row currently being edited.

The add transaction button is restored, and the other buttons are hidden.

The date is reset, input fields are cleared, and any files in the file upload are removed.

```

337 function discardChanges() {
338     document.getElementsByClassName('editing')[0].classList = "bodyRow";
339
340     document.getElementById('add').removeAttribute('hidden');
341     document.getElementById('save').setAttribute('hidden', true);
342     document.getElementById('discard').setAttribute('hidden', true);
343
344     document.getElementById('add').setAttribute('type', 'submit');
345     document.getElementById('save').setAttribute('type', 'button');

```

```

346
347     resetDate();
348     clearInput(true);
349     removeFileUpload();
350 }

```

4.15 sortTable()

The function uses bubble sort to organise rows based on the column number specified by the first argument. The condition evaluated is determined by whether the second argument is set to `true` or `false`.

```

352 function sortTable(column, ascending) {
353     var rows = document.getElementsByClassName('bodyRow');
354
355     var sorting = true;
356     while(sorting) {
357         sorting = false;
358         for(var i = 0; i < (rows.length - 1); i++) {
359             rowA = rows[i].getElementsByTagName('td')[column];
360             rowB = rows[i + 1].getElementsByTagName('td')[column];
361
362             var swap = false;
363
364             if(ascending && rowA.innerHTML.toLowerCase() > rowB.innerHTML.
                 ↪ toLowerCase()) swap = true;
365             else if(!ascending && rowA.innerHTML.toLowerCase() < rowB.
                 ↪ innerHTML.toLowerCase()) swap = true;
366
367             if(swap) {
368                 sorting = true;
369                 document.getElementById('tableBody').insertBefore(rows[i +
                 ↪ 1], rows[i]);
370             }
371         }
372     }
373 }

```

4.16 resetDate()

A new `Date` object is created to get the user's current date. The year, month, and day of this object is organised to use the desired format.

Currently, this format is required to sort the table by date. Changes to the format would require a special function for sorting dates to parse numbers and weight them correctly.

```

375 function resetDate() {
376     const today = new Date();
377     const year = new Intl.DateTimeFormat('en', { year: 'numeric' }).format
                 ↪ (today);
378     const month = new Intl.DateTimeFormat('en', { month: '2-digit' }).
                 ↪ format(today);

```

```

379     const day = new Intl.DateTimeFormat('en', { day: '2-digit' }).format(
380         ↪ today);
381     document.getElementById('date').value = `${year}-${month}-${day}`;
382 }

```

4.17 clearInput()

The type, security, amount, and dollar amount input fields are cleared. If necessary, the account field is also cleared, depending on the value of the argument.

```

384 function clearInput(clearAccount) {
385     if(clearAccount)
386         document.getElementById('account').value = '';
387
388     document.getElementById('type').value = '';
389     document.getElementById('security').value = '';
390     document.getElementById('amount').value = '';
391     document.getElementById('dAmount').value = '';
392 }

```

4.18 validateFilters()

The function takes the filter input fields as arguments and sends each to a function created to validate the specific field. The function returns `false` if any checks fail, and `true` otherwise.

```

394 function validateFilters(filterId, startDate, endDate, filterAccount,
395     ↪ filterType, filterSecurity, minAmount, maxAmount, minDAmount,
396     ↪ maxDAmount, minCostBasis, maxCostBasis) {
397     if(!validateFilterId(filterId)) return false;
398     if(!validateDateRange(startDate, endDate)) return false;
399     if(!validateFilterAccount(filterAccount)) return false;
400     if(!validateFilterSecurity(filterSecurity)) return false;
401     if(!validateAmountRange(minAmount, maxAmount)) return false;
402     if(!validateDAmountRange(minDAmount, maxDAmount)) return false;
403     if(!validateCostBasisRange(minCostBasis, maxCostBasis)) return false;
404     return true;
405 }

```

4.18.1 always true

Some filters require no validation. These are set to always return `true`.

```

406 function validateFilterId(id) {
407     return true;
408 }

```

4.18.2 date range

The function checks that both dates in the range are valid dates. The function then checks that the start date is not greater than the end date.

```
410 function validateDateRange(start, end) {
411     if(!start.checkValidity()) {
412         alert('Error: Invalid Start Date');
413         return false;
414     }
415
416     if(!end.checkValidity()) {
417         alert('Error: Invalid End Date');
418         return false;
419     }
420
421     if(start.valueAsNumber > end.valueAsNumber) {
422         alert('Error: Invalid Date Range');
423         return false;
424     }
425
426     return true;
427 }
```

4.18.3 amount range

The function checks that both amounts are valid numbers. The function then checks that the minimum is not greater than the maximum amount.

```
437 function validateAmountRange(min, max) {
438     if(isNaN(Number(min))) {
439         alert('Error: Min Amount is NaN');
440         return false;
441     }
442
443     if(isNaN(Number(max))) {
444         alert('Error: Max Amount is NaN');
445         return false;
446     }
447
448     if(Number(min) > Number(max) && min != '' && max != '') {
449         alert('Error: Invalid Amount Range');
450         return false;
451     }
452
453     return true;
454 }
```

4.19 stringFilter()

For filters with more complex queries, top level queries are separated by AND operations. These are subdivided by OR operations.

Within an AND block, any one OR must be true for the block to be true. All AND blocks must be true for the query to succeed for the item checked.

```
494 function stringFilter(filtertext, tableitem) {
495     filters = filtertext.split(" && ");
496
497     for(var i = 0; i < filters.length; i++) {
498         filterORs = filters[i].split(" || ");
499         var meetsCriteria = false;
500
501         for(var ii = 0; ii < filterORs.length; ii++) {
502             if(filterORs[ii][0] == "!" && !tableitem.toUpperCase().
                    ↳ includes(filterORs[ii].toUpperCase().substr(1)))
                    ↳ meetsCriteria = true;
503             if(filterORs[ii][0] != "!" && tableitem.toUpperCase().includes
                    ↳ (filterORs[ii].toUpperCase())) meetsCriteria = true;
504         }
505
506         if(!meetsCriteria) return false;
507     }
508
509     return true;
510 }
```

4.20 applyFilter()

If the filter is valid, all previous filters are removed, and each column is filtered as necessary. If any condition fails, the row is hidden. Otherwise, the row remains visible.

```
512 function applyFilter() {
513     unfilterAll();
514
515     rows = document.getElementsByClassName('bodyRow');
516
517     filterId = document.getElementById('filterId').value;
518     startDate = document.getElementById('startDate');
519     endDate = document.getElementById('endDate');
520     filterAccount = document.getElementById('filterAccount').value;
521     filterType = document.getElementById('filterType').value;
522     filterSecurity = document.getElementById('filterSecurity').value;
523     lowAmount = document.getElementById('lowAmount').value;
524     highAmount = document.getElementById('highAmount').value;
525     lowDAmount = document.getElementById('lowDAmount').value;
526     highDAmount = document.getElementById('highDAmount').value;
527     lowCostBasis = document.getElementById('lowCostBasis').value;
528     highCostBasis = document.getElementById('highCostBasis').value;
529
530     if(lowDAmount[0] == '$') lowDAmount = lowDAmount.substr(1);
```



```

531     if(highDAmount[0] == '$') highAmount = highDAmount.substr(1);
532     if(lowCostBasis[0] == '$') lowCostBasis = lowCostBasis.substr(1);
533     if(highCostBasis[0] == '$') highCostBasis = highCostBasis.substr(1);
534
535     if(validateFilters(filterId, startDate, endDate, filterAccount,
        ↪ filterType, filterSecurity, lowAmount, highAmount, lowDAmount,
        ↪ highDAmount, lowCostBasis, highCostBasis)) {
536         for(var i = 0; i < rows.length; i++) {
537             cells = rows[i].getElementsByTagName('td');
538             var hide = false;
539
540             if(filterId != '' && !stringFilter(filterId, cells[0].innerText
        ↪ ))
541                 hide = true;
542
543             if(startDate.value != '' && startDate.value > cells[1].
        ↪ innerText)
544                 hide = true;
545
546             if(endDate.value != '' && endDate.value < cells[1].innerText)
547                 hide = true;
548
549             if(filterAccount != '' && !stringFilter(filterAccount, cells
        ↪ [2].innerText))
550                 hide = true;
551
552             if(filterType != '' && filterType != cells[3].innerText)
553                 hide = true;
554
555             if(filterSecurity != '' && !stringFilter(filterSecurity, cells
        ↪ [4].innerText))
556                 hide = true;
557
558             if(lowAmount != '' && Number(lowAmount) >
        ↪ formattedStringToNumber(cells[5].innerText))
559                 hide = true;
560
561             if(highAmount != '' && Number(highAmount) <
        ↪ formattedStringToNumber(cells[5].innerText))
562                 hide = true;
563
564             if(lowDAmount != '' && Number(lowDAmount) >
        ↪ formattedStringToNumber(cells[6].innerText.substr(1)))
565                 hide = true;
566
567             if(highDAmount != '' && Number(highDAmount) <
        ↪ formattedStringToNumber(cells[6].innerText.substr(1)))
568                 hide = true;
569
570             if(lowCostBasis != '' && Number(lowCostBasis) >
        ↪ formattedStringToNumber(cells[7].innerText.substr(1)))
571                 hide = true;
572
573             if(highCostBasis != '' && Number(highCostBasis) <

```

```

574         ↪ formattedStringToNumber(cells[7].innerText.substr(1)))
575         hide = true;
576         if(filterNA.checked && cells[7].innerText == "N/A")
577             hide = true;
578
579         if(hide)
580             rows[i].setAttribute('hidden', true);
581     }
582 }
583 }

```

4.21 clearFilter()

All previous filters are removed by calling `unfilterAll()`. All filter fields are set to empty strings to clear them. The checkbox is set to false to uncheck it.

```

585 function clearFilter() {
586     unfilterAll();
587
588     var fields = document.getElementsByClassName('filterField');
589
590     for(var i = 0; i < fields.length; i++) {
591         fields[i].value = '';
592     }
593
594     document.getElementById('filterNA').checked = false;
595 }

```

4.22 unfilterAll()

All previous filters are removed by unhiding all body rows.

```

597 function unfilterAll() {
598     rows = document.getElementsByClassName('bodyRow');
599
600     for(var i = 0; i < rows.length; i++) {
601         rows[i].removeAttribute('hidden');
602     }
603 }

```

4.23 toggleID()

Gets the current state of the ID column depending on the text of the button.

If the column needs to be hidden, the `hidden` attribute is added to each ID cell. Frozen column classes are shifted to remove the offset that would be created by the absence of these cells.

If the column needs to be revealed, the `hidden` attribute is removed from each ID cell. Frozen columns are shifted back to create room for the ID column.

```
605 function toggleID() {
606     var button = document.getElementById('toggleId');
607     var rows = document.getElementsByTagName('tr');
608     var cells = rows[0].getElementsByTagName('th');
609
610     if(button.innerText == "Hide Transaction ID") {
611         button.innerText = "Show Transaction ID";
612
613         cells[0].setAttribute('hidden', true);
614         cells[0].classList = "";
615         cells[1].classList = "frozenColumn1";
616         //cells[2].classList = "frozenColumn2";
617         for(var i = 1; i < rows.length; i++) {
618             cells = rows[i].getElementsByTagName('td');
619
620             cells[0].setAttribute('hidden', true);
621             cells[0].classList = "idCell";
622             cells[1].classList = "frozenColumn1";
623             //cells[2].classList = "frozenColumn2";
624         }
625     }
626     else {
627         button.innerText = "Hide Transaction ID";
628
629         cells[0].removeAttribute('hidden');
630         cells[0].classList = "frozenColumn1";
631         cells[1].classList = "frozenColumn2";
632         //cells[2].classList = "frozenColumn3";
633         for(var i = 1; i < rows.length; i++) {
634             cells = rows[i].getElementsByTagName('td');
635
636             cells[0].removeAttribute('hidden');
637             cells[0].classList = "idCell frozenColumn1";
638             cells[1].classList = "frozenColumn2";
639             //cells[2].classList = "frozenColumn3";
640         }
641     }
642 }
```

4.24 readFile()

Gets the contents of a file and sets the value of the `typesArray` element to these contents.

```
644 function readFile(fileIn){
645     if(fileIn.files && fileIn.files[0]) {
646         var reader = new FileReader();
647         reader.onload = function (e) {
648             var output = e.target.result;
649             document.getElementById('typesArray').value = output;
650         };
651     }
```

```

651         reader.readAsText(fileIn.files[0]);
652     }
653 }

```

4.25 saveFile()

The value of the `typesArray` element is encoded for utf-8 and written to a file. The user is prompted to download and save this file.

```

655 function saveFile() {
656     var element = document.createElement('a');
657     element.setAttribute('href', 'data:text/plain;charset=utf-8,' +
        ↪ encodeURIComponent(document.getElementById('typesArray').value));
658     element.setAttribute('download', 'transaction-types.csv');
659
660     element.style.display = 'none';
661     document.body.appendChild(element);
662
663     element.click();
664
665     document.body.removeChild(element);
666 }

```

4.26 applyTypes()

The value of the `typesArray` element is split by commas and saved as an array. This array is passed as an argument to the `setTransactionTypesList()` function.

```

668 function applyTypes() {
669     var typesArray = document.getElementById('typesArray').value.split
        ↪ (' ');
670     setTransactionTypesList(typesArray);
671 }

```

4.27 editTypes()

The current types, which are read by `readCurrentTypes()` are formatted as a comma-separated string and used to fill the `typesArray` element.

```

673 function editTypes() {
674     document.getElementById('typesArray').value = readCurrentTypes().join
        ↪ (' ');
675 }

```

4.28 setTransactionTypes()

The existing lists of transaction types, which are found in the input and filter sections, are cleared. Empty values are added to both to serve as default values. New lists are created from the array passed to this function as an argument. If the type is preceded by an exclamation mark, this is kept in the `value` attribute, but is not displayed to the user.

```
677 function setTransactionTypesList(typesArray) {
678     var type = document.getElementById('type');
679     var filterType = document.getElementById('filterType');
680
681     type.innerHTML = '<option value=""></option>';
682     filterType.innerHTML = '<option value=""></option>';
683
684     for(var i = 0; i < typesArray.length; i++) {
685         var typeAsText = typesArray[i];
686         if(typesArray[i][0] == '!') typeAsText = typesArray[i].substr(1);
687
688         type.innerHTML += '<option value="' + typesArray[i] + '>' +
        ↪ typeAsText + '</option>';
689         filterType.innerHTML += '<option value="' + typeAsText + '>' +
        ↪ typeAsText + '</option>';
690     }
691 }
```

4.29 toggleSection()

The section that the button belongs to is hidden or revealed, depending on the text of the button. The text is updated to reflect the status of the section and the action that can be taken, whether hiding or revealing the section.

```
693 function toggleSection(button) {
694     var form = button.parentElement.parentElement.getElementsByTagName('
        ↪ form')[0];
695
696     if(button.innerText == "Hide") {
697         form.setAttribute("hidden",true);
698         button.innerText = "Show";
699     }
700     else {
701         form.removeAttribute("hidden");
702         button.innerText = "Hide";
703     }
704 }
```

4.30 loadDataLists()

Arrays are constructed from unique strings found in the accounts and securities columns of the table. These are used to create option elements for autofill suggestions in these fields.

```

706 function loadDataLists() {
707     var accountsList = document.getElementById("accountsList");
708     var securitiesList = document.getElementById("securitiesList");
709     var rows = document.getElementsByClassName("bodyRow");
710
711     var accounts = [];
712     var securities = [];
713
714     for(var i = 0; i < rows.length; i++) {
715         var tableAccount = rows[i].getElementsByTagName("td")[2].innerText
716         ↪ ;
717         var tableSecurity = rows[i].getElementsByTagName("td")[4].
718         ↪ innerText;
719
720         if(!accounts.includes(tableAccount)) accounts.push(tableAccount);
721         if(!securities.includes(tableSecurity)) securities.push(
722         ↪ tableSecurity);
723     }
724
725     accountsList.innerHTML = '';
726     securitiesList.innerHTML = '';
727
728     for(var i = 0; i < accounts.length; i++) {
729         accountsList.innerHTML += '<option value="' + accounts[i] + '>';
730     }
731
732     for(var i = 0; i < securities.length; i++) {
733         securitiesList.innerHTML += '<option value="' + securities[i] +
734         ↪ '>';
735     }
736 }

```

4.31 readCurrentTypes()

A list of current types saved as an array, created from the `value` attributes of the options in the input dropdown menu.

```

734 function readCurrentTypes() {
735     var types = document.getElementById('type').getElementsByTagName('
736     ↪ option');
737     var currentTypes = [];
738
739     for(var i = 1; i < types.length; i++) {
740         currentTypes.push(types[i].value);
741     }
742
743     return currentTypes;
744 }

```

4.32 tableToArrays()

The contents of each body row are saved in a two-dimensional array exactly as displayed, except for the file column, which is saved as a list of file names and IDs.

```
745 function tableToArrays() {
746     var rows = document.getElementsByClassName('bodyRow');
747     var data = new Array();
748     data.push(["Transaction Id", "Date", "Account Number", "Transaction
    ↪ Type", "Security", "Amount", "$ Amount", "Cost Basis", "Files"]);
    ↪ ;
749
750     for(var i = 0; i < rows.length; i++) {
751         var cells = rows[i].getElementsByTagName('td');
752         var cellData = new Array();
753
754         for(var j = 0; j < 8; j++) {
755             cellData.push(cells[j].innerText);
756         }
757         cellData.push(getFileNamesIds(cells[8]));
758         data.push(cellData);
759     }
760
761     console.log(data);
762     return data;
763 }
```

4.33 arraysToTable()

All current rows are removed from the table. Buttons in the input field are set to the correct state for adding, as any row being edited would have been removed in the previous step.

For each row stored in the array, the function passes the data to the `addTransactionButton`, starting with the last row saved. This order is necessary as rows are added to the top of the table, so they must be read from the bottom to preserve order.

The row that was just added is popped from the array, allowing the function to proceed to the next row. This loop continues as long as there are rows remaining in the array.

```
765 function arraysToTable(dataArr) {
766     while(document.getElementsByClassName('bodyRow').length > 0) {
767         document.getElementById("tableBody").removeChild(document.
    ↪     getElementsByClassName('bodyRow')[0]);
768     }
769
770     document.getElementById('add').removeAttribute('hidden');
771     document.getElementById('save').setAttribute('hidden', true);
772     document.getElementById('discard').setAttribute('hidden', true);
773
774     document.getElementById('add').setAttribute('type', 'submit');
775     document.getElementById('save').setAttribute('type', 'button');
776
777     while(dataArr.length > 0) {
```

```

778         let data = dataArr[dataArr.length - 1];
779         let files = parseFileNamesIds(data[8]);
780         data.pop();
781         if(files.length > 0) {
782             data.push(files);
783         }
784         addTransaction(data);
785         dataArr.pop();
786     }
787     loadDataLists();
788 }

```

4.34 window.onload = function()

Once the page is loaded, the date is set to the current date and the local database is initialised.

```

790 window.onload = function() {
791     resetDate();
792     initDb();
793 }

```


5 firebaseScript.js

Removes data stored under the **Data** reference and the **Types** reference.

5.1 clearFirebase()

```
1 function clearFirebase() {  
2     firebase.database().ref('Data').remove();  
3     firebase.database().ref('Types').remove();  
4 }
```

5.2 writeToFirebase()

Clears the existing data using `clearFirebase()`. Creates an array from the table and an array containing types. Contents of each array are written to the respective sections of the database.

```
6 function writeToFirebase() {  
7     writeImagesToFirestore("firebase");  
8     clearFirebase();  
9  
10    var data = tableToArray();  
11    var typesArr = readCurrentTypes();  
12  
13    for(var i = 1; i < data.length; i++) {  
14        firebase.database().ref('Data/' + String(i - 1)).set({  
15            id: data[i][0],  
16            date: data[i][1],  
17            account: data[i][2],  
18            type: data[i][3],  
19            security: data[i][4],  
20            amount: data[i][5],  
21            dAmount: data[i][6],  
22            costBasis: data[i][7],  
23            files: data[i][8]  
24        });  
25    }  
26    for(var i = 0; i < typesArr.length; i++) {  
27        firebase.database().ref('Types/' + String(i)).set({  
28            value: typesArr[i]  
29        });  
30    }  
31 }
```

5.3 readFromFirebase()

Takes a snapshot of the database at the time it is being read.

Passes the contents of each row to the `addTransaction()` function.

Creates an array of transaction types from the database, passes this array as an argument to `setTransactionTypesList` \rightarrow `()`.

Account and securities autofill suggestions are updated using `loadDataLists()`.

```
33 function readFromFirebase() {
34     clearIndexedDb("firebase");
35     return firebase.database().ref('/').once('value').then(function(
36          $\rightarrow$  snapshot) {
37
38         while(document.getElementsByClassName('bodyRow').length > 0) {
39             document.getElementById("tableBody").removeChild(document.
40                  $\rightarrow$  getElementsByName('bodyRow')[0]);
41         }
42
43         data = snapshot.val().Data;
44         console.log(data);
45         for(var i = data.length - 1; i >= 0; i--) {
46             let staged = [data[i].id, data[i].date, data[i].account, data[
47                  $\rightarrow$  i].type, data[i].security, data[i].amount, data[i].
48                  $\rightarrow$  dAmount, data[i].costBasis];
49             if(parseFileNamesIds(data[i].files).length > 0) {
50                 staged.push(parseFileNamesIds(data[i].files));
51             }
52             console.log(staged);
53             addTransaction(staged);
54         }
55
56         types = snapshot.val().Types;
57         var typesArr = [];
58         for(var i = 0; i < types.length; i++) {
59             typesArr.push(types[i].value);
60         }
61         setTransactionTypesList(typesArr);
62         loadDataLists();
63     });
64 }
```

5.4 clearFirestore()

The function reads the current Firestore database to get existing document IDs for data and types. Each ID is removed from the database to allow new data to be written.

Note: This function is never called. Instead, it has been used as a template for the `writeToFirestore()`.

```
62 function clearFirestore() {
63     firestore.collection("Data").get().then((querySnapshot) => {
64         querySnapshot.forEach((doc) => {
65             firestore.collection("Data").doc(doc.id).delete();
66         });
67     }).then(function() {
68         firestore.collection("Types").get().then((querySnapshot) => {
69             querySnapshot.forEach((doc) => {
```

```

70         firestore.collection("Types").doc(doc.id).delete();
71     });
72 }
73 }).then(function() { return 0 });
74 }

```

5.5 writeToFirestore()

Uses the template from the `clearFirestore()` function to erase the current database.

When finished, creates documents for each row, containing all the necessary fields to recreate the row. This document contains an index to preserve the order of the original table.

Creates a document for each type, containing the value of the type and an index to store the order in which it should be read.

```

76 function writeToFirestore() {
77     writeImagesToFirestore("firestore");
78     firestore.collection("Data").get().then((querySnapshot) => {
79         querySnapshot.forEach((doc) => {
80             firestore.collection("Data").doc(doc.id).delete();
81         });
82     }).then(function() {
83         firestore.collection("Types").get().then((querySnapshot) => {
84             querySnapshot.forEach((doc) => {
85                 firestore.collection("Types").doc(doc.id).delete();
86             });
87         }).then(function() {
88             var data = tableToArray();
89             var typesArr = readCurrentTypes();
90
91             for(var i = 1; i < data.length; i++) {
92                 firestore.collection("Data").add({
93                     id: data[i][0],
94                     date: data[i][1],
95                     account: data[i][2],
96                     type: data[i][3],
97                     security: data[i][4],
98                     amount: data[i][5],
99                     dAmount: data[i][6],
100                    costBasis: data[i][7],
101                    files: data[i][8],
102                    index: i - 1
103                })
104                .then(function(docRef) {
105                    console.log("Document written with ID: ", docRef.id);
106                })
107                .catch(function(error) {
108                    console.error("Error adding document: ", error);
109                });
110            }
111            for(var i = 0; i < typesArr.length; i++) {
112                firestore.collection("Types").add({

```

```

113         value: typesArr[i],
114         index: i
115     })
116     .then(function(docRef) {
117         console.log("Document written with ID: ", docRef.id);
118     })
119     .catch(function(error) {
120         console.error("Error adding document: ", error);
121     });
122 }
123 })
124 });
125 }

```

5.6 readFromFirestore()

Creates an array of rows, arranged in the original order using the stored `index` property. Creates an array of types using the same method.

Each row is written to the table using the `addTransaction()` function, reading from the last item in the array to preserve the original order. Account and security autofill suggestions are updated by calling `loadDataLists()`.

The types array is passed to `setTransactionTypes()` to update the transaction types.

```

127 function readFromFirestore() {
128     clearIndexedDb("firestore");
129     firestore.collection("Data").get().then((querySnapshot) => {
130         var data = new Array();
131
132         querySnapshot.forEach((doc) => {
133             data[doc.data().index] = doc.data();
134             console.log(data);
135         });
136
137         while(document.getElementsByClassName('bodyRow').length > 0) {
138             document.getElementById("tableBody").removeChild(document.
139                 ↪ getElementsByClassName('bodyRow')[0]);
140         }
141
142         for(let i = data.length - 1; i >= 0; i--) {
143             let staged = [data[i].id, data[i].date, data[i].account, data[
144                 ↪ i].type, data[i].security, data[i].amount, data[i].
145                 ↪ dAmount, data[i].costBasis];
146             if(parseFileNamesIds(data[i].files).length > 0) {
147                 staged.push(parseFileNamesIds(data[i].files));
148             }
149             console.log(staged);
150             addTransaction(staged);
151         }
152         loadDataLists();
153     });
154 }

```

```
152     firestore.collection("Types").get().then((querySnapshot) => {
153         var typesArr = [];
154
155         querySnapshot.forEach((doc) => {
156             typesArr[doc.data().index] = doc.data().value;
157         });
158
159         setTransactionTypesList(typesArr);
160     });
161 }
```

6 googleApiScript.js

6.1 Global Variables

`auth2` is the authentication object created to verify that a user is logged in, and to allow modifications to the sheet on behalf of the user.

`sheetId` and `sheetIdNum` are the IDs for the currently selected sheet, as a numeric value and as a string. Both of these are used to identify the sheet to read from or modify, depending on the situation.

```
1 var auth2;
2
3 var spreadsheetId = "1R0HpaAIUw-JHX8SrzkEPCG1qgI-siJ9oucY6g5e4Co";
4 var sheetId = "Sheet1";
5 var sheetIdNum = 0;
```

6.2 loadSheetData()

The function checks if the user is logged in through OAuth.

If so, `getAllUserSheets()` is called to load the user's Google Sheets.

If not, the function asks them to log in. If successful, it calls `getAllUserSheets()`.

```
7 function loadSheetData() {
8     if(auth2.isSignedIn.get())
9     {
10         getAllUserSheets();
11     }
12     else {
13         authenticate()
14             .then(function() {
15                 if(auth2.isSignedIn.get()) getAllUserSheets();
16             });
17     }
18 }
```

6.3 getNewSheetData()

This function verifies that the user is logged in before getting the tabs from the selected sheet.

If necessary, the function asks the user to log in before reading these tabs.

```
20 function getNewSheetData() {
21     spreadsheetId = document.getElementById('sheet').value;
22     if(auth2.isSignedIn.get())
23     {
24         getTabsOfSheet();
25     }
26     else {
```

```

27         authenticate()
28         .then(function() {
29             if(auth2.isSignedIn.get()) getTabsOfSheet();
30         });
31     }
32 }

```

6.4 populateSheetSelector()

Updates the list of sheets to select from. The first value is always the default sheet. Subsequent values are read from an array passed to this function as an argument.

```

34 function populateSheetSelector(arrayOfSheets) {
35     document.getElementById('sheet').innerHTML = '<option value="1
    ↳ R0HpaAIUw-JHX8SrzvkEPCG1qgI-siJ9oucY6g5e4Co">default</option>';
36
37     for(var i = 0; i < arrayOfSheets.length; i++) {
38         document.getElementById('sheet').innerHTML += '<option value="' +
    ↳ arrayOfSheets[i].id + '">' + arrayOfSheets[i].name + '</
    ↳ option>';
39     }
40 }

```

6.5 getNewTabData()

Gets the string and numeric IDs of the currently selected tab and stores them as global variables.

```

42 function getNewTabData() {
43     data = document.getElementById('tab').value.split(/,(.+)/);
44     sheetIdNum = data[0];
45     sheetId = data[1];
46 }

```

6.6 populateTabSelector()

Takes an array of tabs as an argument. Creates a new option for each element in the array, allowing the user to select any tab from the sheet.

```

48 function populateTabSelector(arrayOfTabs) {
49     document.getElementById('tab').innerHTML = '';
50
51     for(var i = 0; i < arrayOfTabs.length; i++) {
52         document.getElementById('tab').innerHTML += '<option value="' +
    ↳ arrayOfTabs[i].properties.sheetId + ',' + arrayOfTabs[i].
    ↳ properties.title + '">' + arrayOfTabs[i].properties.title +
    ↳ '</option>';
53     }
54 }

```

6.7 getAllUserSheets()

Gets the name and ID of 1000 of the user's sheets, sorted by name.

```
56 function getAllUserSheets() {
57     return gapi.client.drive.files.list({
58         "pageSize": 1000,
59         "orderBy": "name",
60         "q": "mimeType = 'application/vnd.google-apps.spreadsheet'",
61     })
62     .then(function(response) {
63         populateSheetSelector(JSON.parse(response.body).files);
64         getNewSheetData();
65         console.log("Response", response);
66     },
67     function(err) { console.error("Execute error", err); });
68 }
```

6.8 getTabsOfSheet()

Gets JSON data for all tabs of the selected sheet. Populates the dropdown menu for tabs and gets necessary data for the tab, including the numeric and string IDs.

```
70 function getTabsOfSheet() {
71     return gapi.client.sheets.spreadsheets.get({
72         "spreadsheetId": spreadsheetId,
73         "includeGridData": false
74     })
75     .then(function(response) {
76         populateTabSelector(JSON.parse(response.body).sheets);
77         getNewTabData();
78         console.log("Response", response);
79     },
80     function(err) { console.error("Execute error", err); });
81 }
```

6.9 authenticate()

Attempts to sign the user in through Google using OAuth.

```
83 function authenticate() {
84     return gapi.auth2.getAuthInstance()
85         .signIn({scope: "https://www.googleapis.com/auth/drive"})
86         .then(function() { console.log("Sign-in successful"); },
87         function(err) { console.error("Error signing in", err); });
88 }
```


6.10 loadClientSheets()

Loads the sheets API.

```
90 function loadClientSheets() {
91   gapi.client.setApiKey("AIzaSyDC6JNuMW78Q-gWsp0PFEaTsICYjHWymAo");
92   return gapi.client.load("https://content.googleapis.com/discovery/v1/apis/
    ↪ sheets/v4/rest")
93     .then(function() { console.log("GAPI client loaded for API");
    ↪ loadSheetData(); },
94       function(err) { console.error("Error loading GAPI client for API",
    ↪ err); });
95 }
```

6.11 loadClient()

Loads the drive API.

```
97 function loadClient() {
98   gapi.client.setApiKey("AIzaSyDC6JNuMW78Q-gWsp0PFEaTsICYjHWymAo");
99   return gapi.client.load("https://content.googleapis.com/discovery/v1/apis/
    ↪ drive/v3/rest")
100     .then(function() { console.log("GAPI client loaded for API");
    ↪ loadClientSheets(); },
101       function(err) { console.error("Error loading GAPI client for API",
    ↪ err); });
102 }
```

6.12 readGoogleSheetDB()

Reads the table section of the sheet and stores values as a two-dimensional array. Passes this array as an argument to `arraysToTable()` to write the arrays to the website. Calls `readGoogleTypes()` to get transaction types from the sheet.

```
104 function readGoogleSheetDB() {
105   clearIndexedDb("sheets");
106   return gapi.client.sheets.spreadsheets.values.get({
107     "spreadsheetId": spreadsheetId,
108     "range": sheetId + "!A2:I214748354"
109   })
110     .then(function(response) {
111       console.log("Response", response);
112
113       dataArr = [];
114       if(JSON.parse(response.body).values != undefined) {
115         dataArr = JSON.parse(response.body).values;
116       }
117       arraysToTable(dataArr);
118
119       readGoogleTypes();
120     });
121 }
```

```

120     },
121     function(err) { console.error("Execute error", err); });
122 }

```

6.13 readGoogleTypes()


The column of transaction types is read as an array, using columns as the major dimension. The result is passed to `setTransactionTypesList`.

```

124 function readGoogleTypes() {
125     return gapi.client.sheets.spreadsheets.values.get({
126         "spreadsheetId": spreadsheetId,
127         "range": sheetId + "!J1:J214748354",
128         "majorDimension": "COLUMNS"
129     })
130 }
131     .then(function(response) {
132         console.log("Response", response);
133         var typesArr = JSON.parse(response.body).values[0];
134         setTransactionTypesList(typesArr);
135     },
136     function(err) { console.error("Execute error", err); });
137 }

```

6.14 writeGoogleSheetDB()

If the user is authenticated, the function begins the process of saving to the database by calling `setGoogleRows`  (). Otherwise, the function attempts to authenticate the user, and starts this process if successful.

```

139 function writeGoogleSheetDB() {
140     if(auth2.isSignedIn.get())
141     {
142         setGoogleRows()
143     }
144     else {
145         authenticate()
146         .then(function() {
147             if(auth2.isSignedIn.get()) setGoogleRows();
148         });
149     }
150 }

```

6.15 setGoogleRows()

The rows and columns of the spreadsheet are reduced to the minimum amount, removing most residual data and preventing the sheet from growing over time.

```

152 function setGoogleRows() {
153     return gapi.client.sheets.spreadsheets.batchUpdate({

```

```

154     "spreadsheetId": spreadsheetId,
155     "resource": {
156       "requests": [
157         {
158           "updateSheetProperties": {
159             "properties": {
160               "gridProperties": {
161                 "columnCount": 10,
162                 "rowCount": 1
163               },
164               "sheetId": sheetIdNum
165             },
166             "fields": "gridProperties"
167           }
168         }
169       ]
170     }
171   })
172   .then(function(response) {
173     console.log("Response", response);
174
175     clearGoogleRow();
176   },
177   function(err) { console.error("Execute error", err); });
178 }

```

6.16 clearGoogleRow()

The remaining row is cleared to remove data from previous use.

```

180 function clearGoogleRow() {
181   return gapi.client.sheets.spreadsheets.values.clear({
182     "spreadsheetId": spreadsheetId,
183     "range": sheetId + "!A1:J1",
184     "resource": {}
185   })
186   .then(function(response) {
187     console.log("Response", response);
188
189     writeGoogleDB();
190   },
191   function(err) { console.error("Execute error", err); });
192 }

```

6.17 writeGoogleDB()

`tableToArray()` is used to create a two-dimensional array from the table, which is easily mapped to the spreadsheet. The array of transaction types is written similarly, using columns as the major dimension to write them vertically. The spreadsheet will automatically expand as necessary during this process.

```

194 function writeGoogleDB() {
195     writeImagesToFirestore("sheets");
196     return gapi.client.sheets.spreadsheets.values.batchUpdate({
197         "spreadsheetId": spreadsheetId,
198         "resource": {
199             "data": [
200                 {
201                     "range": sheetId + "!A1",
202                     "values": tableToArrays(),
203                     "majorDimension": "ROWS"
204                 },
205                 {
206                     "range": sheetId + "!J1",
207                     "values": [readCurrentTypes()],
208                     "majorDimension": "COLUMNS"
209                 }
210             ],
211             "valueInputOption": "RAW"
212         }
213     })
214     .then(function(response) {
215         console.log("Response", response);
216     },
217     function(err) { console.error("Execute error", err); });
218 }

```

6.18 gapi.load()

Starts the OAuth plugin using the client ID from the Google Developer Console for this project. Calls `loadClient()` to begin loading API clients.

```

220 gapi.load("client:auth2", function() {
221     auth2 = gapi.auth2.init({client_id: "217251662395-9
    ↪ pu2qa1hubgrblav1nhvnfaasc6povv.apps.googleusercontent.com"});
222     loadClient();
223 });

```

7 imageFirestore.js

7.1 writeImagesToFirestore()

Reads the database to get all document IDs of images for the dataset, then deletes those files.

Loads data from the local database. Creates a new document on the Firestore database for each file, saved with the file ID, type, name, and base64 encoded data.

```
1 function writeImagesToFirestore(database) {
2   firestore.collection("Images:" + database).get().then((querySnapshot)
3     ↪ => {
4     querySnapshot.forEach((doc) => {
5       firestore.collection("Images:" + database).doc(doc.id).delete
6         ↪ ();
7     });
8   }).then(function() {
9     var trans = db.transaction(['files'], 'readonly');
10    var dlReq = trans.objectStore('files').getAll();
11
12    dlReq.onerror = function(e) {
13      console.log('error reading data');
14      console.error(e);
15    };
16
17    dlReq.onsuccess = function(e) {
18      console.log(dlReq.result);
19
20      for(let i = 0; i < dlReq.result.length; i++) {
21        firestore.collection("Images:" + database).add({
22          id: dlReq.result[i].id,
23          type: dlReq.result[i].type,
24          name: dlReq.result[i].name,
25          data: dlReq.result[i].data,
26        })
27        .then(function(docRef) {
28          console.log("Image written with ID: ", docRef.id);
29        })
30        .catch(function(error) {
31          console.log("Error adding image: ", error);
32        });
33      }
34    };
35  });
36 }
```

7.2 readImagesFromFirestore()

Reads files from Firestore for a specific database. Saves these files in the local database.

```
36 function readImagesFromFirestore(database) {
```

```

37     firestore.collection("Images:" + database).get().then((querySnapshot)
    ↪ => {
38         let trans = db.transaction(['files'], 'readwrite');
39
40         trans.oncomplete = function(e) {
41             console.log('data stored');
42         }
43
44         querySnapshot.forEach((doc) => {
45             console.log("Writing ", doc.data().name);
46
47             let ob = {
48                 id: doc.data().id,
49                 type: doc.data().type,
50                 name: doc.data().name,
51                 data: doc.data().data
52             };
53
54             let addReq = trans.objectStore('files').put(ob);
55         });
56     });
57 }

```

7.3 getFileNamesIds()

Creates a slash-delimited string from the file names and IDs.

```

59 function getFileNamesIds(cell) {
60     var links = cell.getElementsByTagName('a');
61     var result = '';
62
63     for(let i = 0; i < links.length; i++) {
64         result += links[i].innerText + '/' + links[i].getAttribute('
    ↪ onclick').split('')[1] + '/';
65     }
66
67     return result;
68 }

```

7.4 parseFileNamesIds()

Takes a string and splits it by '/' characters. Groups characters in pairs and saves this information in a two-dimensional array.

```

70 function parseFileNamesIds(string) {
71     var result = new Array();
72
73     if(string != '') {
74         let strarr = string.split('/');
75         for(let i = 0; i < strarr.length - 1; i += 2) {

```

```
76         result.push([strarr[i + 1], strarr[i + 0]]);
77     }
78 }
79
80 return result;
81 }
```

7.5 clearIndexedDb()

Removes all data from the local database and calls the function to read from Firestore.

```
83 function clearIndexedDb(database) {
84     console.log("db reset");
85     let trans = db.transaction(['files'], 'readwrite');
86     var clearReq = trans.objectStore('files').clear();
87
88     trans.oncomplete = function(e) {
89         readImagesFromFirestore(database);
90     }
91 }
```

8 localStorageScript.js

8.1 Global Variables

Variables to store the database object, the version number, and the state of the database.

```
1 let db;
2 let dbVersion = 1;
3 let dbReady = false;
4
5 var fileEditted = false;
```

8.2 initDb()

Deletes the database if it previously existed by calling `indexedDB.deleteDatabase()` to clear it from previous use.

Once the old database has been deleted, a new database is created with a name `'FileStorage'` and a version number determined by the `dbVersion` global variable.

The global variable `db` is set to the database object for future reference. An object store is created using `db.createObjectStore()`, called `files`, which is keyed using the `'id'` field.

Lastly, `dbReady` is set to `true` to signal to other functions that the database can be used for reading and writing.

```
7 function initDb() {
8     let reset = indexedDB.deleteDatabase('FileStorage');
9     reset.onsuccess = function(a) {
10         let request = indexedDB.open('FileStorage', dbVersion);
11
12         request.onerror = function(e) {
13             console.error('Unable to open database.');
```


8.3 fileUploadChanged()

This function is called whenever the user performs an action that would change the contents of the file upload in the input section.

In this case, the function sets the label to display the number of files present in the element. The global variable `fileEditted` is set to `true` so that the app will know that the row has to be updated if the user is editing a row.

```
29 function fileUploadChanged() {
30     fileIn = document.getElementById('fileUpload');
31     if(fileIn.files && fileIn.files[0]) {
32         document.getElementById('fileUploadLabel').innerHTML = String(
33             ↪ fileIn.files.length) + " file(s)";
34     }
35     fileEditted = true;
36     console.log("updated file upload");
37 }
```

8.4 uploadFile()

Calls the `fileIdGenerator()` function to create a unique ID to reference the file being uploaded.

If a file exists in the file upload at the index specified, the function reads the file as a binary string and converts to base64 for storage.

The file creates an object with the file ID, type, name, and base64 encoded data. A transaction is created with a request to add the object to the local database. Because the key has been set to use the `id` property, the ID will automatically use the ID from the object.

Once the transaction is complete, the new ID and file name is added to an array. The index is incremented to prepare for the next file, and the function calls itself with this new index.

Once the function reaches an index it cannot read (i.e. the function has reached the end of the list), the file upload element is cleared and the array of IDs and names is added to the `data` array. The function then passes this argument to the callback function, which is specified as an argument of this function. The callback function is likely the function responsible for adding or editing transactions.

```
38 function uploadFile(data, cb, fileList, index) {
39     fileId = fileIdGenerator();
40     fileIn = document.getElementById('fileUpload');
41     if(fileIn.files && fileIn.files[index]) {
42         var reader = new FileReader();
43         reader.onload = function (e) {
44             console.log(e.target.result);
45
46             let bits = btoa(e.target.result);
47             let ob = {
48                 id: fileId,
49                 type: fileIn.files[index].type,
50                 name: fileIn.files[index].name,
51                 data: bits
52             };
53         };
54     }
55 }
```

```

53
54         let trans = db.transaction(['files'], 'readwrite');
55         let addReq = trans.objectStore('files').put(ob);
56
57         addReq.onerror = function(e) {
58             console.log('error storing data');
59             console.error(e);
60         }
61
62         trans.oncomplete = function(e) {
63             console.log('data stored');
64             fileList.push([fileId, fileIn.files[index].name]);
65             index++;
66             uploadFile(data, cb, fileList, index);
67         }
68     };
69     reader.readAsBinaryString(fileIn.files[index])
70 }
71 else {
72     removeFileUpload();
73     data.push(fileList);
74     cb(data);
75 }
76 }

```

8.5 addFile()

This function is responsible for appending files to an existing list.

Similar to the `uploadFile()` function, an initial index is provided and the function reads through each index until it reaches one where there is not file, indicating the end of the list.

The method for reading and sending files to the local database is identical to the `uploadFile()` function. Files are stored as an object with an ID, type, name, and base64 encoded data. A transaction is created with a request to add this object to the database.

Once added, the function gets the parent element of the file input element and adds a new row including the file, with a hyperlink to download the file and a button to remove it. The index is then incremented and the function calls itself, attempting to read the next file in the list.

```

78 function addFile(fileIn, index) {
79     fileId = fileIdGenerator();
80     if(fileIn.files && fileIn.files[index]) {
81         var reader = new FileReader();
82         reader.onload = function (e) {
83             console.log(e.target.result);
84
85             let bits = btoa(e.target.result);
86             let ob = {
87                 id: fileId,
88                 type: fileIn.files[index].type,
89                 name: fileIn.files[index].name,
90                 data: bits

```

```

91         };
92
93         let trans = db.transaction(['files'], 'readwrite');
94         let addReq = trans.objectStore('files').put(ob);
95
96         addReq.onerror = function(e) {
97             console.log('error storing data');
98             console.error(e);
99         }
100
101         trans.oncomplete = function(e) {
102             console.log('data stored');
103             let table = fileIn.parentElement.getElementsByTagName('
104                 ↳ tbody')[0];
105             let newRowContent = "<tr><td><a onclick='downloadFile(\" +
106                 ↳ fileId + \"');' href='javascript:void(0);'>" +
107                 ↳ fileIn.files[index].name + "</a></td>";
108             newRowContent += "<td><button type='button' onclick='
109                 ↳ removeFileFromTable(\" + fileId + \"', this);'>-</
110                 ↳ button></td></tr>";
111             table.innerHTML += newRowContent;
112             index++;
113             addFile(fileIn, index);
114         }
115     };
116     reader.readAsBinaryString(fileIn.files[index])
117 }
118 else {
119     fileIn.value = null;
120 }
121 }

```

8.6 updateExistingFileName()

Note: The name of this function was from when the app only handled a single file per row. The purpose of this function is to edit all files in a row. The name is no longer accurate, however, it was kept to simplify the upgrade.

This function is used to edit the files stored in a row when the editing a row and the file upload has been changed, indicating that the existing files should be overwritten.

The function loops through the list of files for that row and gets the ID of each file, using the `onclick` attribute of the anchor tags. Each of these IDs is passed to the `deleteFileFromIndexedDB()` function to have them removed.

A string `fileContent` is created to store the contents of the list of files as its being constructed.

Each file is added to this element with the proper HTML formatting using a loop. Each entry contains the correct name and file ID.

Once all files have been added, a new file input element is appended to the bottom of the list to allow new files to be easily uploaded. The contents of the cell are then replaced by this `fileContent` string to update the UI.

```

118 function updateExistingFileName(data) {
119     for(let i = 0; i < data[0].getElementsByName('tr').length; i++) {
120         let fileId = data[0].getElementsByName('tr')[i].
            ↳ getElementsByName('a')[0].getAttribute('onclick').split
            ↳ (','')[1];
121         deleteFileFromIndexedDB(fileId);
122     }
123
124     var fileContent = '<table><tbody>';
125     if(data.length > 1) {
126         for(let i = 0; i < data[1].length; i++) {
127             fileContent += "<tr><td><a onclick='downloadFile(\" + data[1][
                ↳ i][0] + \"');' href='javascript:void(0);'>" + data[1][i
                ↳ ][1] + "</a></td>";
128             fileContent += "<td><button type='button' onclick='
                ↳ removeFileFromTable(\" + data[1][i][0] + \"', this);'>-</
                ↳ button></td></tr>";
129         }
130     }
131     fileContent += '</tbody></table><input type="file" onchange="addFile(
        ↳ this, 0);" multiple/>';
132     data[0].innerHTML = fileContent;
133 }

```

8.7 removeFileFromTable()

When an individual file is being removed, this function gets the ID of the file as an argument and passes it to the `deleteFileFromIndexedDB()` function to have it removed from the local database. The row element that contains the link to this file is also removed.

```

135 function removeFileFromTable(fileId, cell) {
136     var row = cell.parentElement.parentElement;
137
138     if(confirm("Delete " + row.getElementsByName('a')[0].innerText +
        ↳ " ?")) {
139         row.parentElement.removeChild(row);
140         deleteFileFromIndexedDB(fileId);
141     }
142 }

```

8.8 deleteFileFromIndexedDB()

Given a file ID to remove, this function creates a transaction to the local database, with a request to delete the entry at the given ID.

```

144 function deleteFileFromIndexedDB(fileId) {
145     let trans = db.transaction(['files'], 'readwrite');
146     let addReq = trans.objectStore('files').delete(fileId);
147 }

```

8.9 removeFileUpload()

This function is responsible for clearing the file upload in the input, which is done when the **x** button is pressed, when a transaction is added, or when a row is edited.

The function sets the file upload to `null`, effectively clearing it.

The label for the file upload is set to `Upload file`, to indicate that no file currently exists in the element.

The `fileEdited` variable is set to `true` to indicate that a change has been made and that, if this process is an edit, a row has to be updated.

```
149 function removeFileUpload() {
150     document.getElementById('fileUpload').value = null;
151     document.getElementById('fileUploadLabel').innerHTML = "Upload file";
152     fileEdited = true;
153 }
```

8.10 downloadFile()

A `readonly` transaction is created with a request to read the object at the specified ID.

If the request is successful, an anchor element is created with the attributes to download the base64 encoded data as a file. The type of this file is set using the `type` property of the object being read. The name of the file is set using the `name` property.

The anchor element is set to not display and is added to the HTML page. A click on this element is simulated, which prompts the user to download the file. The anchor element is then removed.

```
155 function downloadFile(fileId) {
156     console.log('downloading');
157     var trans = db.transaction(['files'], 'readonly');
158     var dlReq = trans.objectStore('files').get(fileId);
159
160     dlReq.onerror = function(e) {
161         console.log('error reading data');
162         console.error(e);
163     };
164
165     dlReq.onsuccess = function(e) {
166         console.log('data read');
167         console.log(dlReq.result);
168         var element = document.createElement('a');
169         element.setAttribute('href', 'data:' + dlReq.result.type + ';'
170             ↪ + 'base64,' + dlReq.result.data);
171         element.setAttribute('download', dlReq.result.name);
172
173         element.style.display = 'none';
174         document.body.appendChild(element);
175
176         element.click();
177
178         document.body.removeChild(element);
179     };
180 }
```

```
178     };  
179 }
```

8.11 window.onbeforeunload = function()

This function is meant to execute before the page exits. The purpose of this function is to remove the local database.

```
1 window.onbeforeunload = function(){  
2     indexedDB.deleteDatabase('FileStorage');  
3 }
```

9 mysqlScript.js

9.1 writeToMySQL()

This function is responsible for initiating the request to update the MySQL database. It makes a POST request to the API endpoint. It sends the table as an array and the transaction types as an array, formatted as JSON data.


```
1 function writeToMySQL() {
2     writeImagesToFirestore("mysql");
3     var data = tableToArray();
4     var types = readCurrentTypes();
5
6     fetch('http://localhost:5000/api', {
7         method: 'POST',
8         headers: {
9             'Content-Type': 'application/json',
10        },
11        body: JSON.stringify([data, types]),
12    });
13 }
```

9.2 readFromMySQL()


This function reads data from the MySQL database and updates the HTML table.

A GET request is made to the API endpoint, which expects to receive JSON data. The data is parsed into an array, containing the table data and the transaction types.

The current HTML table is cleared by removing all body rows.

The table data is used to recreate the table from the MySQL database by sending each row to the `addTransaction`  `()` function.

The transaction types are updated by sending the array to `setTransactionTypesList()`.

```
15 function readFromMySQL() {
16     clearIndexedDb("mysql");
17     fetch('http://localhost:5000/api')
18         .then(response => {
19             return response.json()
20         })
21         .then(fullresponse => {
22             console.log(fullresponse);
23
24             while(document.getElementsByClassName('bodyRow').length > 0) {
25                 document.getElementById("tableBody").removeChild(document.
26                      getElementsByName('bodyRow')[0]);
27             }
28
29             var data = fullresponse[0];
30             for(var i = data.length - 1; i >= 0; i--) {
```

```

30         let staged = [data[i].id, data[i].date, data[i].account,
31             ↪ data[i].type, data[i].security, data[i].amount, data
32             ↪ [i].dAmount, data[i].costBasis];
33         if(parseFileNamesIds(data[i].files).length > 0) {
34             staged.push(parseFileNamesIds(data[i].files));
35         }
36         addTransaction(staged);
37     }
38     loadDataLists();
39
40     var types = fullresponse[1];
41     var typesArr = [];
42     for(var i = 0; i < types.length; i++) {
43         typesArr.push(types[i].typename);
44     }
45     setTransactionTypesList(typesArr);
46 }

```


10 NodeJs

10.1 Dependencies

```
1 const functions = require('firebase-functions');
2 const express = require('express');
3 const mysql = require('mysql');
4 const cors = require('cors');
5 const api = express();
6 api.use(cors({ origin: true }));
```

10.2 mysql.createConnection()

```
8 var con = mysql.createConnection({
9   host: "localhost",
10  user: "user",
11  password: "pass",
12  database: "mydb"
13 });
```

10.3 con.connect()

```
15 con.connect(function(err) {
16   if (err) throw err;
17   console.log("Connected to MySQL Database!");
18 });
```

10.4 readDataFromMySQL()

```
20 function readDataFromMySQL(cb) {
21   con.query("SELECT * FROM data;", function (err, data, fields) {
22     if (err) throw err;
23     cb(data);
24   });
25 }
```

10.5 readTypesFromMySQL()

```
27 function readTypesFromMySQL(cb) {
28   con.query("SELECT * FROM types;", function (err, types, fields) {
29     if (err) throw err;
30     cb(types);
31   });
32 }
```

10.6 writeToMySQL()

```
34 function writeToMySQL(jsonData) {
35     var data = jsonData[0];
36     var types = jsonData[1];
37
38     con.query("TRUNCATE TABLE data;");
39
40     for(var i = 1; i < data.length; i++) {
41         con.query("INSERT INTO data (id, date, account, type, security,
42             ↪ amount, dAmount, costBasis, files) VALUES ('" + data[i].join
43             ↪ ("', '" + "');");
44     }
45
46     con.query("TRUNCATE TABLE types;");
47
48     for(var i = 0; i < types.length; i++) {
49         con.query("INSERT INTO types (typename) VALUES ('" + types[i] +
50             ↪ "');");
51     }
52 }
```

10.7 api.post()

```
51 api.post('/api', (req, res) => {
52     console.log(req.body);
53     writeToMySQL(req.body);
54     return res.status(200).send(req.body);
55 });
```

10.8 api.get()

```
57 api.get('/api', (req, res) => {
58     readDataFromMySQL(function(data) {
59         console.log(data);
60         readTypesFromMySQL(function(types) {
61             console.log(types);
62             return res.status(200).send([ data, types ]);
63         });
64     });
65 });
```

10.9 exports.api

```
67 exports.api = functions.https.onRequest(api);
```

11 CSS

11.1 Vertical Scrolling Table

11.2 Horizontal Scrolling on Overflow

11.3 Miscellaneous

11.3.1 Sort buttons

11.3.2 Editing highlight

11.3.3 Table borders