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.

```
<!DOCTYPE html>
```

3.1.2 meta charset

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

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

3.1.3 link rel="stylesheet"

Imports the CSS file.

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

3.1.4 Scripts

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

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

Imports the Google API library.

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

Imports other Javascript files, responsible for database management.

```
<script src="./googleApiScript.js"></script>
<script src="./mysqlScript.js"></script>
<script src="./localStorageScript.js"></script>
<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.

```
<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.

```
<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.

```
<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.

```
<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.

```
<label for="type">Transaction Type:</label><br/>
<select id="type" name="type">
    <option value=""></option>
    <option value="BUY">BUY</option>
    <option value="SELL">SELL</option>
    <option value="!DIVIDEND">DIVIDEND</option>
    <option value="!INTEREST">INTEREST</option>
    <option value="!WITHDRAW">WITHDRAW</option>
    <option value="!DEPOSIT">DEPOSIT</option>
</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.

```
    <label id="fileUploadLabel" for="fileUpload">Upload file</label>
    <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.

```
<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.

```
<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.

```
<label for="filterNa">Filter N/A:</label>
<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>
```

```

<script>
// Your web app's Firebase configuration
var firebaseConfig = {
    apiKey: "AIzaSyAmZLFZHDAB9evhvNunxOe5GxXRd_OizmU",
    authDomain: "financial-transactions-6f065.firebaseapp.com",
    databaseURL: "https://financial-transactions-6f065.firebaseio.com",
    projectId: "financial-transactions-6f065",
    storageBucket: "financial-transactions-6f065.appspot.com",
    messagingSenderId: "82206982479",
    appId: "1:82206982479:web:8937bbd1bd4fb6022b053a",
    measurementId: "G-0564DT8RNQ"
};
// Initialize Firebase
firebase.initializeApp(firebaseConfig);
firebase.analytics();

var database = firebase.database();
var firestore = firebase.firestore();
</script>

<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.

```
function formattedStringToNumber(numberAsString) {
    var number;

    if(numberAsString[0] == '$') {
        numberAsString = numberAsString.substr(1);
    }

    number = Number(numberAsString.replace(/,/g, ''));

    return number;
}
```

4.2 numberToFormattedString()

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

```
function numberToFormattedString(number) {
    var numberAsString;

    numberAsString = String(number).replace(/\B(?=(\d{3})+(?!\d))/g, ",");

    return numberAsString;
}
```

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.

```
function getData() {
    var date = document.getElementById("date");
    var account = document.getElementById("account").value;
    var type = document.getElementById("type").value;
    var security = document.getElementById("security").value;
    var amount = document.getElementById("amount").value;
    var dAmount = document.getElementById("dAmount").value;

    security = security.toUpperCase();

    amount = formattedStringToNumber(amount);
```

```

    dAmount = formattedStringToNumber(dAmount);

    if(validate(date, account, type, security, amount, dAmount)) {
        var costBasis = '$' + numberToFormattedString(calculateCostBasis(amount, dAmount));
        date = date.value;

        amount = numberToFormattedString(amount);
        dAmount = '$' + numberToFormattedString(dAmount.toFixed(2));

        return [ date, account, type, security, amount, dAmount, costBasis ];
    }
    else return false;
}
```

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`.

```
function validate(date, account, type, security, amount, dAmount) {
    if(!validateDate(date)) return false;
    if(!validateAccount(account)) return false;
    if(!validateType(type)) return false;
    if(!validateSecurity(security)) return false;
    if(!validateAmount(amount)) return false;
    if(!validateDAmount(dAmount)) return false;

    return true;
}
```

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`.

```
function validateAccount(account) {
    if(account == '') {
        alert('Error: Missing Account Number');
        return false;
    }

    return true;
}
```

4.4.2 Check NaN

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

```
function validateAmount(amount) {
    if(amount == '') {
        alert('Error: Missing Amount');
        return false;
    }

    if(isNaN(amount)) {
        alert('Error: Invalid Amount');
        return false;
    }

    return true;
}
```

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.

```
function validateDate(date) {
    realDate = new Date();
    inputDate = date.valueAsNumber;

    if(date.value == '') {
        alert('Error: Missing date');
        return false;
    }

    if(!date.checkValidity()) {
        alert('Error: Invalid date');
        return false;
    }

    if(realDate.valueOf() < inputDate) {
        alert('Error: Date is in the future');
        return false;
    }

    return true;
}
```

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.

```
function generateId() {
    var id = '';
    var idLength = 6;

    var characters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789';
    var charactersLength = characters.length;

    var unique = false;

    while(!unique) {
        for(var i = 0; i < idLength; i++) {
            id += characters.charAt(Math.floor(Math.random() * charactersLength));
        }

        unique = true;
        for(var i = 0; i < document.getElementsByClassName('idCell').length; i++) {
            if(document.getElementsByClassName('idCell')[i].innerText == id) {
                unique = false;
                break;
            }
        }
    }
    return id;
}
```

4.6 calculateCostBasis()

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

```
function calculateCostBasis(amount, dAmount) {
    costBasis = (dAmount / amount).toFixed(2);
    return costBasis;
}
```

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.

```
function addTransaction(data) {
    var staging = data;
    var tableBody = document.getElementById('tableBody');
    var newRow = tableBody.insertRow(0);
    newRow.classList += "bodyRow";

```

```
    var actionsContent = "<button type='button' onclick='editRow(this)'>Edit</button><button type='button' onclick='deleteRow(this)'>Delete</button>";
    var fileContent = '<table><tbody>';
    if(data.length > 8) {
        for(let i = 0; i < data[8].length; i++) {
            fileContent += "<tr><td><a onclick='downloadFile(`" + data[8][i][0] + "`);' href='javascript:void(0);'>" + data[8][i][1] + "</a></td>";
            fileContent += "<td><button type='button' onclick='removeFileFromTable(`" + data[8][i][0] + "`, this);'>-</button></td></tr>";
        }
    }
    fileContent += '</tbody></table><input type="file" onchange="addFile(this, 0);" multiple/>';
    staging[8] = fileContent;
    staging[9] = actionsContent;

    var calculateCostBasis = true;
    if(data[3][0] == '!') {
        calculateCostBasis = false;
        data[3] = data[3].substr(1);
    }

    for(var i = 0; i < 10; i++) {
        var newCell = newRow.insertCell(i);
        var idShowing = (document.getElementById('toggleId').innerText == "Hide Transaction ID");

        newCell.innerHTML = data[i];

        if(i == 0) {
            if(idShowing)
                newCell.classList = "idCell frozenColumn1";
            else {
                newCell.classList = "idCell";
                newCell.setAttribute("hidden", true);
            }
        }
        else if(i == 1) {
            if(idShowing)
                newCell.classList = "frozenColumn2";
            else
                newCell.classList = "frozenColumn1";
        }
        /*
        else if(i == 2) {
            if(idShowing)
                newCell.classList = "frozenColumn3";
            else
                newCell.classList = "frozenColumn2";
        }
        */
```

```

        if(i == 7 && !calculateCostBasis) {
            newCell.innerHTML = "N/A";
        }
    }
}
```

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.

```
function fileIdGenerator() {
    return Math.floor(Math.random() *
        ↪ 1000000000000000000000000000000000000000000000).toString(36);
}
```

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.

```
function addTransactionButton() {
    var data = getData();
    if(data) {
        var id = generateId();
        data.unshift(id);

        fileList = new Array()
        uploadFile(data, addTransactionWithFileName, fileList, 0);
        clearInput(false);
    }
}
```

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.

```
function addTransactionWithFileName(data) {
    addTransaction(data);
    console.log(data);
    loadDataLists();
}
```

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).

```
function deleteRow(button) {
    var row = button.parentElement.parentElement;

    var fileRows = row.getElementsByTagName('td')[8].getElementsByTagName
        ↪ ('table')[0].getElementsByTagName('tr');
    for(let i = 0; i < fileRows.length; i++) {
        let fileId = fileRows[i].getElementsByTagName('a')[0].getAttribute
            ↪ ('onclick').split('`')[1];
        deleteFileFromIndexedDB(fileId);
    }

    document.getElementById("tableBody").removeChild(row);

    if(document.getElementsByClassName('editing').length == 0) {
        document.getElementById('add').removeAttribute('hidden');
        document.getElementById('save').setAttribute('hidden', true);
        document.getElementById('discard').setAttribute('hidden', true);

        document.getElementById('add').setAttribute('type','submit');
        document.getElementById('save').setAttribute('type','button');
    }
    loadDataLists();
}
```

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.

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

```

```
    var row = button.parentElement.parentElement;
    var rowContent = row.getElementsByTagName('td');
    row.classList = "bodyRow editing";

    document.getElementById('date').value = rowContent[1].innerText;
    document.getElementById('account').value = rowContent[2].innerText;

    document.getElementById('type').value = rowContent[3].innerText;
    if(document.getElementById('type').value == '') document.getElementById('type').value = '!' + rowContent[3].innerText;

    document.getElementById('security').value = rowContent[4].innerText;
    document.getElementById('amount').value = rowContent[5].innerText;
    document.getElementById('dAmount').value = rowContent[6].innerText;

    document.getElementById('add').setAttribute('hidden', true);
    document.getElementById('save').removeAttribute('hidden');
    document.getElementById('discard').removeAttribute('hidden');

    document.getElementById('add').setAttribute('type','button');
    document.getElementById('save').setAttribute('type','submit');

    removeFileUpload();
    uploadLabel = document.getElementById('fileUploadLabel');
    if(rowContent[8].getElementsByTagName('tr').length > 0) {
        uploadLabel.innerHTML = String(rowContent[8].getElementsByTagName('tr').length) + " file(s)";
    }
    fileEditted = false;
}
```

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.

```
function saveChanges() {
```

```
    data = getData();
    if(data) {
        rowToEdit = document.getElementsByClassName('editing')[0];
        cellsToEdit = rowToEdit.getElementsByTagName('td');

        if(data[2][0] == '!') {
            data[2] = data[2].substr(1);
            data[6] = "N/A";
        }

        for(var i = 0; i < data.length; i++) {
            cellsToEdit[i + 1].innerHTML = data[i];
        }
        rowToEdit.classList = "bodyRow";

        document.getElementById('add').removeAttribute('hidden');
        document.getElementById('save').setAttribute('hidden', true);
        document.getElementById('discard').setAttribute('hidden', true);

        document.getElementById('add').setAttribute('type','submit');
        document.getElementById('save').setAttribute('type','button');

        if(fileEditted) {
            uploadFile([cellsToEdit[8]], updateExistingFileName, new Array
                ↪ (), 0);
        }
        else {
            removeFileUpload();
        }

        resetDate();
        clearInput(true);
        loadDataLists();
    }
}
```

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.

```
function discardChanges() {
    document.getElementsByClassName('editing')[0].classList = "bodyRow";

    document.getElementById('add').removeAttribute('hidden');
    document.getElementById('save').setAttribute('hidden', true);
    document.getElementById('discard').setAttribute('hidden', true);

    document.getElementById('add').setAttribute('type','submit');
    document.getElementById('save').setAttribute('type','button');
```

```

    resetDate();
    clearInput(true);
    removeFileUpload();
}
```

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`.

```
function sortTable(column, ascending) {
    var rows = document.getElementsByClassName('bodyRow');

    var sorting = true;
    while(sorting) {
        sorting = false;
        for(var i = 0; i < (rows.length - 1); i++) {
            rowA = rows[i].getElementsByTagName('td')[column];
            rowB = rows[i + 1].getElementsByTagName('td')[column];

            var swap = false;

            if(ascending && rowA.innerHTML.toLowerCase() > rowB.innerHTML.toLowerCase()) swap = true;
            else if(!ascending && rowA.innerHTML.toLowerCase() < rowB.innerHTML.toLowerCase()) swap = true;

            if(swap) {
                sorting = true;
                document.getElementById('tableBody').insertBefore(rows[i + 1], rows[i]);
            }
        }
    }
}
```

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.

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

```
    const day = new Intl.DateTimeFormat('en', { day: '2-digit' }).format(today);

    document.getElementById('date').value = `${year}-${month}-${day}`;
}
```

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.

```
function clearInput(clearAccount) {
    if(clearAccount)
        document.getElementById('account').value = '';

    document.getElementById('type').value = '';
    document.getElementById('security').value = '';
    document.getElementById('amount').value = '';
    document.getElementById('dAmount').value = '';
}
```

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.

```
function validateFilters(filterId, startDate, endDate, filterAccount, filterType, filterSecurity, minAmount, maxAmount, minDAmount, maxDAmount, minCostBasis, maxCostBasis) {
    if(!validateFilterId(filterId)) return false;
    if(!validateDateRange(startDate, endDate)) return false;
    if(!validateFilterAccount(filterAccount)) return false;
    if(!validateFilterSecurity(filterSecurity)) return false;
    if(!validateAmountRange(minAmount, maxAmount)) return false;
    if(!validateDAmountRange(minDAmount, maxDAmount)) return false;
    if(!validateCostBasisRange(minCostBasis, maxCostBasis)) return false;

    return true;
}
```

4.18.1 always true

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

```
function validateFilterId(id) {
    return true;
}
```

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.

```
function validateDateRange(start, end) {
    if(!start.checkValidity()) {
        alert('Error: Invalid Start Date');
        return false;
    }

    if(!end.checkValidity()) {
        alert('Error: Invalid End Date');
        return false;
    }

    if(start.valueAsNumber > end.valueAsNumber) {
        alert('Error: Invalid Date Range');
        return false;
    }

    return true;
}
```

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.

```
function validateAmountRange(min, max) {
    if(isNaN(Number(min))) {
        alert('Error: Min Amount is NaN');
        return false;
    }

    if(isNaN(Number(max))) {
        alert('Error: Max Amount is NaN');
        return false;
    }

    if(Number(min) > Number(max) && min != '' && max != '') {
        alert('Error: Invalid Amount Range');
        return false;
    }

    return true;
}
```

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.

```
function stringFilter(filtertext, tableitem) {
    filters = filtertext.split(" && ");

    for(var i = 0; i < filters.length; i++) {
        filterORs = filters[i].split(" || ");
        var meetsCriteria = false;

        for(var ii = 0; ii < filterORs.length; ii++) {
            if(filterORs[ii][0] == "!" && !tableitem.toUpperCase().includes(filterORs[ii].toUpperCase().substr(1))) meetsCriteria = true;
            if(filterORs[ii][0] != "!" && tableitem.toUpperCase().includes(filterORs[ii].toUpperCase())) meetsCriteria = true;
        }

        if(!meetsCriteria) return false;
    }

    return true;
}
```

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.

```
function applyFilter() {
    unfilterAll();

    rows = document.getElementsByClassName('bodyRow');

    filterId = document.getElementById('filterId').value;
    startDate = document.getElementById('startDate');
    endDate = document.getElementById('endDate');
    filterAccount = document.getElementById('filterAccount').value;
    filterType = document.getElementById('filterType').value;
    filterSecurity = document.getElementById('filterSecurity').value;
    lowAmount = document.getElementById('lowAmount').value;
    highAmount = document.getElementById('highAmount').value;
    lowDAmount = document.getElementById('lowDAmount').value;
    highDAmount = document.getElementById('highDAmount').value;
    lowCostBasis = document.getElementById('lowCostBasis').value;
    highCostBasis = document.getElementById('highCostBasis').value;

    if(lowDAmount[0] == '$') lowDAmount = lowDAmount.substr(1);
```

```
    if(highDAmount[0] == '$') highAmount = highDAmount.substr(1);
    if(lowCostBasis[0] == '$') lowCostBasis = lowCostBasis.substr(1);
    if(highCostBasis[0] == '$') highCostBasis = highCostBasis.substr(1);

    if(validateFilters(filterId, startDate, endDate, filterAccount, filterType, filterSecurity, lowAmount, highAmount, lowDAmount, highDAmount, lowCostBasis, highCostBasis)) {
        for(var i = 0; i < rows.length; i++) {
            cells = rows[i].getElementsByTagName('td');
            var hide = false;

            if(filterId != '' && !stringFilter(filterId,cells[0].innerText))
                hide = true;

            if(startDate.value != '' && startDate.value > cells[1].innerText)
                hide = true;

            if(endDate.value != '' && endDate.value < cells[1].innerText)
                hide = true;

            if(filterAccount != '' && !stringFilter(filterAccount, cells[2].innerText))
                hide = true;

            if(filterType != '' && filterType != cells[3].innerText)
                hide = true;

            if(filterSecurity != '' && !stringFilter(filterSecurity, cells[4].innerText))
                hide = true;

            if(lowAmount != '' && Number(lowAmount) > formattedStringToNumber(cells[5].innerText))
                hide = true;

            if(highAmount != '' && Number(highAmount) < formattedStringToNumber(cells[5].innerText))
                hide = true;

            if(lowDAmount != '' && Number(lowDAmount) > formattedStringToNumber(cells[6].innerText.substr(1)))
                hide = true;

            if(highDAmount != '' && Number(highDAmount) < formattedStringToNumber(cells[6].innerText.substr(1)))
                hide = true;

            if(lowCostBasis != '' && Number(lowCostBasis) > formattedStringToNumber(cells[7].innerText.substr(1)))
                hide = true;

            if(highCostBasis != '' && Number(highCostBasis) <
```

```
                ↪ formattedStringToNumber(cells[7].innerText.substr(1)))
                hide = true;

            if(filterNA.checked && cells[7].innerText == "N/A")
                hide = true;

            if(hide)
                rows[i].setAttribute('hidden', true);
        }
    }
}
```

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.

```
function clearFilter() {
    unfilterAll();

    var fields = document.getElementsByClassName('filterField');

    for(var i = 0; i < fields.length; i++) {
        fields[i].value = '';
    }

    document.getElementById('filterNA').checked = false;
}
```

4.22 unfilterAll()

All previous filters are removed by unhiding all body rows.

```
function unfilterAll() {
    rows = document.getElementsByClassName('bodyRow');

    for(var i = 0; i < rows.length; i++) {
        rows[i].removeAttribute('hidden');
    }
}
```

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.

```
function toggleID() {
    var button = document.getElementById('toggleId');
    var rows = document.getElementsByTagName('tr');
    var cells = rows[0].getElementsByTagName('th');

    if(button.innerText == "Hide Transaction ID") {
        button.innerText = "Show Transaction ID";

        cells[0].setAttribute('hidden', true);
        cells[0].classList = "";
        cells[1].classList = "frozenColumn1";
        //cells[2].classList = "frozenColumn2";
        for(var i = 1; i < rows.length; i++) {
            cells = rows[i].getElementsByTagName('td');

            cells[0].setAttribute('hidden', true);
            cells[0].classList = "idCell";
            cells[1].classList = "frozenColumn1";
            //cells[2].classList = "frozenColumn2";
        }
    }
    else {
        button.innerText = "Hide Transaction ID";

        cells[0].removeAttribute('hidden');
        cells[0].classList = "frozenColumn1";
        cells[1].classList = "frozenColumn2";
        //cells[2].classList = "frozenColumn3";
        for(var i = 1; i < rows.length; i++) {
            cells = rows[i].getElementsByTagName('td');

            cells[0].removeAttribute('hidden');
            cells[0].classList = "idCell frozenColumn1";
            cells[1].classList = "frozenColumn2";
            //cells[2].classList = "frozenColumn3";
        }
    }
}
```

4.24 readFile()

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

```
function readFile(fileIn){
    if(fileIn.files && fileIn.files[0]) {
        var reader = new FileReader();
        reader.onload = function (e) {
            var output = e.target.result;
            document.getElementById('typesArray').value = output;
        };
```

```
        reader.readAsText(fileIn.files[0]);
    }
}
```

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.

```
function saveFile() {
  var element = document.createElement('a');
  element.setAttribute('href', 'data:text/plain;charset=utf-8,' +
     ↪ encodeURIComponent(document.getElementById('typesArray').value));
  element.setAttribute('download', 'transaction-types.csv');

  element.style.display = 'none';
  document.body.appendChild(element);

  element.click();

  document.body.removeChild(element);
}
```

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.

```
function applyTypes() {
    var typesArray = document.getElementById('typesArray').value.split
       ↪ (',');
    setTransactionTypesList(typesArray);
}
```

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.

```
function editTypes() {
    document.getElementById('typesArray').value = readCurrentTypes().join
       ↪ (',');
}
```

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 preceeded by an exclamation mark, this is kept in the `value` attribute, but is not displayed to the user.

```
function setTransactionTypesList(typesArray) {
    var type = document.getElementById('type');
    var filterType = document.getElementById('filterType');

    type.innerHTML = '<option value=""></option>';
    filterType.innerHTML = '<option value=""></option>';

    for(var i = 0; i < typesArray.length; i++) {
        var typeAsText = typesArray[i];
        if(typesArray[i][0] == '!') typeAsText = typesArray[i].substr(1);

        type.innerHTML += '<option value="' + typesArray[i] + '">' +
            ↪ typeAsText + '</option>';
        filterType.innerHTML += '<option value="' + typeAsText + '">' +
            ↪ typeAsText + '</option>';
    }
}
```

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.

```
function toggleSection(button) {
    var form = button.parentElement.parentElement.getElementsByTagName('
        ↪ form')[0];

    if(button.innerText == "Hide") {
        form.setAttribute("hidden",true);
        button.innerText = "Show";
    }
    else {
        form.removeAttribute("hidden");
        button.innerText = "Hide";
    }
}
```

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.

```
function loadDataLists() {
    var accountsList = document.getElementById("accountsList");
    var securitiesList = document.getElementById("securitiesList");
    var rows = document.getElementsByClassName("bodyRow");

    var accounts = [];
    var securities = [];

    for(var i = 0; i < rows.length; i++) {
        var tableAccount = rows[i].getElementsByTagName("td")[2].innerText;
        var tableSecurity = rows[i].getElementsByTagName("td")[4].innerText;

        if(!accounts.includes(tableAccount)) accounts.push(tableAccount);
        if(!securities.includes(tableSecurity)) securities.push(tableSecurity);
    }

    accountsList.innerHTML = '';
    securitiesList.innerHTML = '';

    for(var i = 0; i < accounts.length; i++) {
        accountsList.innerHTML += '<option value="' + accounts[i] + '"/>';
    }

    for(var i = 0; i < securities.length; i++) {
        securitiesList.innerHTML += '<option value="' + securities[i] + '"/>';
    }
}
```

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.

```
function readCurrentTypes() {
    var types = document.getElementById('type').getElementsByTagName('option');
    var currentTypes = [];

    for(var i = 1; i < types.length; i++) {
        currentTypes.push(types[i].value);
    }

    return currentTypes;
}
```

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.

```
function tableToArrays() {
    var rows = document.getElementsByClassName('bodyRow');
    var data = new Array();
    data.push(["Transaction Id", "Date", "Account Number", "Transaction Type", "Security", "Amount", "$ Amount", "Cost Basis", "Files"]);

    for(var i = 0; i < rows.length; i++) {
        var cells = rows[i].getElementsByTagName('td');
        var cellData = new Array();

        for(var j = 0; j < 8; j++) {
            cellData.push(cells[j].innerText);
        }
        cellData.push(getFileNamesIds(cells[8]));
        data.push(cellData);
    }

    console.log(data);
    return data;
}
```

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.

```
function arraysToTable(dataArr) {
    while(document.getElementsByClassName('bodyRow').length > 0) {
        document.getElementById("tableBody").removeChild(document.getElementsByClassName('bodyRow')[0]);
    }

    document.getElementById('add').removeAttribute('hidden');
    document.getElementById('save').setAttribute('hidden', true);
    document.getElementById('discard').setAttribute('hidden', true);

    document.getElementById('add').setAttribute('type', 'submit');
    document.getElementById('save').setAttribute('type', 'button');

    while(dataArr.length > 0) {
```

```
        let data = dataArr[dataArr.length - 1];
        let files = parseFileNamesIds(data[8]);
        data.pop();
        if(files.length > 0) {
            data.push(files);
        }
        addTransaction(data);
        dataArr.pop();
    }
    loadDataLists();
}
```

4.34 window.onload = function()

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

```
window.onload = function() {
    resetDate();
    initDb();
}
```

5 firebaseScript.js

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

5.1 clearFirebase()

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

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.

```
function writeToFirebase() {
    writeImagesToFirestore("firebase");
    clearFirebase();

    var data = tableToArrays();
    var typesArr = readCurrentTypes();

    for(var i = 1; i < data.length; i++) {
        firebase.database().ref('Data/' + String(i - 1)).set({
            id: data[i][0],
            date: data[i][1],
            account: data[i][2],
            type: data[i][3],
            security: data[i][4],
            amount: data[i][5],
            dAmount: data[i][6],
            costBasis: data[i][7],
            files: data[i][8]
        });
    }
    for(var i = 0; i < typesArr.length; i++) {
        firebase.database().ref('Types/' + String(i)).set({
            value: typesArr[i]
        });
    }
}
```

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()`.

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

```
function readFromFirebase() {
    clearIndexedDb("firebase");
    return firebase.database().ref('/').once('value').then(function(snapshot) {

        while(document.getElementsByClassName('bodyRow').length > 0) {
            document.getElementById("tableBody").removeChild(document.getElementsByClassName('bodyRow')[0]);
        }

        data = snapshot.val().Data;
        console.log(data);
        for(var i = data.length - 1; i >= 0; i--) {
            let staged = [data[i].id, data[i].date, data[i].account, data[i].type, data[i].security, data[i].amount, data[i].dAmount, data[i].costBasis];
            if(parseFileNamesIds(data[i].files).length > 0) {
                staged.push(parseFileNamesIds(data[i].files));
            }
            console.log(staged);
            addTransaction(staged);
        }

        types = snapshot.val().Types;
        var typesArr = [];
        for(var i = 0; i < types.length; i++) {
            typesArr.push(types[i].value);
        }
        setTransactionTypesList(typesArr);
        loadDataLists();
    });
}
```

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()`.

```
function clearFirestore() {
    firestore.collection("Data").get().then((querySnapshot) => {
        querySnapshot.forEach((doc) => {
            firestore.collection("Data").doc(doc.id).delete();
        });
    }).then(function() {
        firestore.collection("Types").get().then((querySnapshot) => {
            querySnapshot.forEach((doc) => {
```

```
                firestore.collection("Types").doc(doc.id).delete();
            });
        })
    }).then(function() { return 0 });
}
```

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.

```
function writeToFirestore() {
    writeImagesToFirestore("firestore");
    firestore.collection("Data").get().then((querySnapshot) => {
        querySnapshot.forEach((doc) => {
            firestore.collection("Data").doc(doc.id).delete();
        });
    }).then(function() {
        firestore.collection("Types").get().then((querySnapshot) => {
            querySnapshot.forEach((doc) => {
                firestore.collection("Types").doc(doc.id).delete();
            });
        }).then(function() {
            var data = tableToArrays();
            var typesArr = readCurrentTypes();

            for(var i = 1; i < data.length; i++) {
                firestore.collection("Data").add({
                    id: data[i][0],
                    date: data[i][1],
                    account: data[i][2],
                    type: data[i][3],
                    security: data[i][4],
                    amount: data[i][5],
                    dAmount: data[i][6],
                    costBasis: data[i][7],
                    files: data[i][8],
                    index: i - 1
                })
                .then(function(docRef) {
                    console.log("Document written with ID: ", docRef.id);
                })
                .catch(function(error) {
                    console.error("Error adding document: ", error);
                });
            }
            for(var i = 0; i < typesArr.length; i++) {
                firestore.collection("Types").add({
```

```
                    value: typesArr[i],
                    index: i
                })
                .then(function(docRef) {
                    console.log("Document written with ID: ", docRef.id);
                })
                .catch(function(error) {
                    console.error("Error adding document: ", error);
                });
            }
        })
    });
}
```

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.

```
function readFromFirestore() {
    clearIndexedDb("firestore");
    firestore.collection("Data").get().then((querySnapshot) => {
        var data = new Array();

        querySnapshot.forEach((doc) => {
            data[doc.data().index] = doc.data();
            console.log(data);
        });

        while(document.getElementsByClassName('bodyRow').length > 0) {
            document.getElementById("tableBody").removeChild(document.
                ↪ getElementsByClassName('bodyRow')[0]);
        }

        for(let i = data.length - 1; i >= 0; i--) {
            let staged = [data[i].id, data[i].date, data[i].account, data[
                ↪ i].type, data[i].security, data[i].amount, data[i].
                ↪ dAmount, data[i].costBasis];
            if(parseFileNamesIds(data[i].files).length > 0) {
                staged.push(parseFileNamesIds(data[i].files));
            }
            console.log(staged);
            addTransaction(staged);
        }
        loadDataLists();
    });

```

```
    firestore.collection("Types").get().then((querySnapshot) => {
        var typesArr = [];

        querySnapshot.forEach((doc) => {
            typesArr[doc.data().index] = doc.data().value;
        });

        setTransactionTypesList(typesArr);
    });
}
```

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 `shettIdNum` 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.

```
var auth2;

var spreadsheetId = "1R0HpaAIUw-JHX8SrzvkEPCG1qgI-siJ9oucY6g5e4Co";
var sheetId = "Sheet1";
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()`.

```
function loadSheetData() {
    if(auth2.isSignedIn.get())
    {
        getAllUserSheets();
    }
    else {
        authenticate()
            .then(function() {
                if(auth2.isSignedIn.get()) getAllUserSheets();
            });
    }
}
```

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.

```
function getNewSheetData() {
    spreadsheetId = document.getElementById('sheet').value;
    if(auth2.isSignedIn.get())
    {
        getTabsOfSheet();
    }
    else {
```

```
        authenticate()
            .then(function() {
                if(auth2.isSignedIn.get()) getTabsOfSheet();
            });
    }
}
```

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.

```
function populateSheetSelector(arrayOfSheets) {
    document.getElementById('sheet').innerHTML = '<option value="1R0HpaAIUw-JHX8SrzvkEPCG1qgI-siJ9oucY6g5e4Co">default</option>';

    for(var i = 0; i < arrayOfSheets.length; i++) {
        document.getElementById('sheet').innerHTML += '<option value="' + arrayOfSheets[i].id + '">' + arrayOfSheets[i].name + '</option>';
    }
}
```

6.5 getNewTabData()

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

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

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.

```
function populateTabSelector(arrayOfTabs) {
    document.getElementById('tab').innerHTML = '';

    for(var i = 0; i < arrayOfTabs.length; i++) {
        document.getElementById('tab').innerHTML += '<option value="' + arrayOfTabs[i].properties.sheetId + ',' + arrayOfTabs[i].properties.title + '">' + arrayOfTabs[i].properties.title + '</option>';
    }
}
```

6.7 getAllUserSheets()

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

```
function getAllUserSheets() {
    return gapi.client.drive.files.list({
        "pageSize": 1000,
        "orderBy": "name",
        "q": "mimeType = 'application/vnd.google-apps.spreadsheet'",
    })
        .then(function(response) {
            populateSheetSelector(JSON.parse(response.body).files);
            getNewSheetData();
            console.log("Response", response);
        },
        function(err) { console.error("Execute error", err); });
}
```

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.

```
function getTabsOfSheet() {
    return gapi.client.sheets.spreadsheets.get({
      "spreadsheetId": spreadsheetId,
      "includeGridData": false
    })
        .then(function(response) {
            populateTabSelector(JSON.parse(response.body).sheets);
            getNewTabData();
            console.log("Response", response);
        },
        function(err) { console.error("Execute error", err); });
}
```

6.9 authenticate()

Attempts to sign the user in through Google using OAuth.

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

6.10 loadClientSheets()

Loads the sheets API.

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

6.11 loadClient()

Loads the drive API.

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

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.

```
function readGoogleSheetDB() {
    clearIndexedDb("sheets");
    return gapi.client.sheets.spreadsheets.values.get({
        "spreadsheetId": spreadsheetId,
        "range": sheetId + "!A2:I214748354"
    })
        .then(function(response) {
            console.log("Response", response);

            dataArr = [];
            if(JSON.parse(response.body).values != undefined) {
                dataArr = JSON.parse(response.body).values;
            }
            arraysToTable(dataArr);

            readGoogleTypes();
```

```
        },
        function(err) { console.error("Execute error", err); });
}
```

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`.

```
function readGoogleTypes() {
    return gapi.client.sheets.spreadsheets.values.get({
        "spreadsheetId": spreadsheetId,
        "range": sheetId + "!J1:J214748354",
        "majorDimension": "COLUMNS"

    })
        .then(function(response) {
            console.log("Response", response);
            var typesArr = JSON.parse(response.body).values[0];
            setTransactionTypesList(typesArr);
        },
        function(err) { console.error("Execute error", err); });
}
```

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.

```
function writeGoogleSheetDB() {
    if(auth2.isSignedIn.get())
    {
        setGoogleRows()
    }
    else {
        authenticate()
            .then(function() {
                if(auth2.isSignedIn.get()) setGoogleRows();
            });
    }
}
```

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.

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

```
        "spreadsheetId": spreadsheetId,
        "resource": {
        "requests": [
            {
            "updateSheetProperties": {
                "properties": {
                    "gridProperties": {
                        "columnCount": 10,
                        "rowCount": 1
                },
                    "sheetId": sheetIdNum
                },
                "fields": "gridProperties"
            }
            }
        ]
        }
    })
        .then(function(response) {
            console.log("Response", response);

            clearGoogleRow();
        },
        function(err) { console.error("Execute error", err); });
}
```

6.16 clearGoogleRow()

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

```
function clearGoogleRow() {
    return gapi.client.sheets.spreadsheets.values.clear({
    "spreadsheetId": spreadsheetId,
    "range": sheetId + "!A1:J1",
    "resource": {}
    })
        .then(function(response) {
            console.log("Response", response);

            writeGoogleDB();
        },
        function(err) { console.error("Execute error", err); });
}
```

6.17 writeGoogleDB()

`tableToArrays()` 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.

```
function writeGoogleDB() {
    writeImagesToFirestore("sheets");
    return gapi.client.sheets.spreadsheets.values.batchUpdate({
        "spreadsheetId": spreadsheetId,
        "resource": {
            "data": [
            {
                "range": sheetId + "!A1",
                "values": tableToArrays(),
                "majorDimension": "ROWS"
            },
            {
                "range": sheetId + "!J1",
                "values": [readCurrentTypes()],
                "majorDimension": "COLUMNS"
            }
            ],
            "valueInputOption": "RAW"
        }
        })
        .then(function(response) {
            console.log("Response", response);
        },
        function(err) { console.error("Execute error", err); });
}
```

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.

```
gapi.load("client:auth2", function() {
    auth2 = gapi.auth2.init({client_id: "217251662395-9
        ↪ pu2qa1hubgrblav1nhvnfaascd6povv.apps.googleusercontent.com"});
    loadClient();
});
```

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.

```
function writeImagesToFirestore(database) {
    firestore.collection("Images:" + database).get().then((querySnapshot) => {
        querySnapshot.forEach((doc) => {
            firestore.collection("Images:" + database).doc(doc.id).delete();
        });
    }).then(function() {
        var trans = db.transaction(['files'], 'readonly');
        var dlReq = trans.objectStore('files').getAll();

        dlReq.onerror = function(e) {
            console.log('error reading data');
            console.error(e);
        };

        dlReq.onsuccess = function(e) {
            console.log(dlReq.result);

            for(let i = 0; i < dlReq.result.length; i++) {
                firestore.collection("Images:" + database).add({
                    id: dlReq.result[i].id,
                    type: dlReq.result[i].type,
                    name: dlReq.result[i].name,
                    data: dlReq.result[i].data,
                })
                .then(function(docRef) {
                    console.log("Image written with ID: ", docRef.id);
                })
                .catch(function(error) {
                    console.log("Error adding image: ", error);
                });
            }
        };
    });
}
```

7.2 readImagesFromFirestore()

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

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

```
    firestore.collection("Images:" + database).get().then((querySnapshot) => {
        let trans = db.transaction(['files'], 'readwrite');

        trans.oncomplete = function(e) {
            console.log('data stored');
        }

        querySnapshot.forEach((doc) => {
            console.log("Writing ", doc.data().name);

            let ob = {
                id: doc.data().id,
                type: doc.data().type,
                name: doc.data().name,
                data: doc.data().data
            };

            let addReq = trans.objectStore('files').put(ob);
        });
    });
}
```

7.3 getFileNamesIds()

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

```
function getFileNamesIds(cell) {
    var links = cell.getElementsByTagName('a');
    var result = '';

    for(let i = 0; i < links.length; i++) {
        result += links[i].innerText + '/' + links[i].getAttribute('onclick').split('`')[1] + '/';
    }

    return result;
}
```

7.4 parseFileNamesIds()

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

```
function parseFileNamesIds(string) {
    var result = new Array();

    if(string != '') {
        let strarr = string.split('/');
        for(let i = 0; i < strarr.length - 1; i += 2) {
```

```
            result.push([strarr[i + 1], strarr[i + 0]]);
        }
    }

    return result;
}
```

7.5 clearIndexedDb()

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

```
function clearIndexedDb(database) {
    console.log("db reset");
    let trans = db.transaction(['files'], 'readwrite');
    var clearReq = trans.objectStore('files').clear();

    trans.oncomplete = function(e) {
        readImagesFromFirestore(database);
    }
}
```

8 localStorageScript.js

8.1 Global Variables

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

```
let db;
let dbVersion = 1;
let dbReady = false;

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.

```
function initDb() {
    let reset = indexedDB.deleteDatabase('FileStorage');
    reset.onsuccess = function(a) {
        let request = indexedDB.open('FileStorage', dbVersion);

        request.onerror = function(e) {
            console.error('Unable to open database.');
        }

        request.onsuccess = function(e) {
            db = e.target.result;
            console.log('db opened');
        }

        request.onupgradeneeded = function(e) {
            let db = e.target.result;
            db.createObjectStore('files', {keyPath:'id', autoIncrement:
                ↪ false});
            dbReady = true;
        }
    }
}
```

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.

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

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.

```
function uploadFile(data, cb, fileList, index) {
    fileId = fileIdGenerator();
    fileIn = document.getElementById('fileUpload');
    if(fileIn.files && fileIn.files[index]) {
        var reader = new FileReader();
        reader.onload = function (e) {
            console.log(e.target.result);

            let bits = btoa(e.target.result);
            let ob = {
                id: fileId,
                type: fileIn.files[index].type,
                name: fileIn.files[index].name,
                data: bits
            };
```

```

            let trans = db.transaction(['files'], 'readwrite');
            let addReq = trans.objectStore('files').put(ob);

            addReq.onerror = function(e) {
                console.log('error storing data');
                console.error(e);
            }

            trans.oncomplete = function(e) {
                console.log('data stored');
                fileList.push([fileId, fileIn.files[index].name]);
                index++;
                uploadFile(data, cb, fileList, index);
            }
        };
        reader.readAsBinaryString(fileIn.files[index])
    }
    else {
        removeFileUpload();
        data.push(fileList);
        cb(data);
    }
}
```

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.

```
function addFile(fileIn, index) {
    fileId = fileIdGenerator();
    if(fileIn.files && fileIn.files[index]) {
        var reader = new FileReader();
        reader.onload = function (e) {
            console.log(e.target.result);

            let bits = btoa(e.target.result);
            let ob = {
                id: fileId,
                type: fileIn.files[index].type,
                name: fileIn.files[index].name,
                data: bits
```

```
            };

            let trans = db.transaction(['files'], 'readwrite');
            let addReq = trans.objectStore('files').put(ob);

            addReq.onerror = function(e) {
                console.log('error storing data');
                console.error(e);
            }

            trans.oncomplete = function(e) {
                console.log('data stored');
                let table = fileIn.parentElement.getElementsByTagName('tbody')[0];
                let newRowContent = "<tr><td><a onclick='downloadFile(`" + fileId + "`);' href='javascript:void(0);'>" + fileIn.files[index].name + "</a></td>";
                newRowContent += "<td><button type='button' onclick='removeFileFromTable(`" + fileId + "`, this);'>-</button></td></tr>";
                table.innerHTML += newRowContent;
                index++;
                addFile(fileIn, index);
            }
        };
        reader.readAsBinaryString(fileIn.files[index])
    }
    else {
        fileIn.value = null;
    }
}
```

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.

```
function updateExistingFileName(data) {
    for(let i = 0; i < data[0].getElementsByTagName('tr').length; i++) {
        let fileId = data[0].getElementsByTagName('tr')[i].getElementsByTagName('a')[0].getAttribute('onclick').split('`')[1];
        deleteFileFromIndexedDB(fileId);
    }

    var fileContent = '<table><tbody>';
    if(data.length > 1) {
        for(let i = 0; i < data[1].length; i++) {
            fileContent += "<tr><td><a onclick='downloadFile(`" + data[1][i][0] + "`);' href='javascript:void(0);'>" + data[1][i][1] + "</a></td>";
            fileContent += "<td><button type='button' onclick='removeFileFromTable(`" + data[1][i][0] + "`, this);'>-</button></td></tr>";
        }
    }
    fileContent += '</tbody></table><input type="file" onchange="addFile(this, 0);" multiple/>';
    data[0].innerHTML = fileContent;
}
```

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.

```
function removeFileFromTable(fileId, cell) {
    var row = cell.parentElement.parentElement;

    if(confirm("Delete " + row.getElementsByTagName('a')[0].innerText + "?")) {
        row.parentElement.removeChild(row);
        deleteFileFromIndexedDB(fileId);
    }
}
```

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.

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

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 `fileEditted` 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.

```
function removeFileUpload() {
    document.getElementById('fileUpload').value = null;
    document.getElementById('fileUploadLabel').innerHTML = "Upload file";
    fileEditted = true;
}
```

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.

```
function downloadFile(fileId) {
    console.log('downloading');
    var trans = db.transaction(['files'], 'readonly');
    var dlReq = trans.objectStore('files').get(fileId);

    dlReq.onerror = function(e) {
        console.log('error reading data');
        console.error(e);
    };

    dlReq.onsuccess = function(e) {
        console.log('data read');
        console.log(dlReq.result);
        var element = document.createElement('a');
        element.setAttribute('href', 'data:' + dlReq.result.type + ';
            ↪ base64,' + dlReq.result.data);
        element.setAttribute('download', dlReq.result.name);

        element.style.display = 'none';
        document.body.appendChild(element);

        element.click();

        document.body.removeChild(element);
```

```
    };
}
```

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.

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

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.

```
function writeToMySQL() {
    writeImagesToFirestore("mysql");
    var data = tableToArrays();
    var types = readCurrentTypes();

    fetch('http://localhost:5000/api', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
        },
        body: JSON.stringify([data, types]),
    });
}
```

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 use 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()`.

```
function readFromMySQL() {
    clearIndexedDb("mysql");
    fetch('http://localhost:5000/api')
        .then(response => {
            return response.json()
        })
        .then(fullresponse => {
            console.log(fullresponse);

            while(document.getElementsByClassName('bodyRow').length > 0) {
                document.getElementById("tableBody").removeChild(document.getElementsByClassName('bodyRow')[0]);
            }

            var data = fullresponse[0];
            for(var i = data.length - 1; i >= 0; i--) {
```

```
                let staged = [data[i].id, data[i].date, data[i].account, data[i].type, data[i].security, data[i].amount, data[i].dAmount, data[i].costBasis];
                if(parseFileNamesIds(data[i].files).length > 0) {
                    staged.push(parseFileNamesIds(data[i].files));
                }
                addTransaction(staged);
            }
            loadDataLists();

            var types = fullresponse[1];
            var typesArr = [];
            for(var i = 0; i < types.length; i++) {
                typesArr.push(types[i].typename);
            }
            setTransactionTypesList(typesArr);
        })
}
```

10 NodeJs

10.1 Dependencies

`firebase-functions` is required to create Cloud Functions and set up triggers.

`express` is used to route requests to appropriate endpoints.

`mysql` is used to interact with the MySQL database.

`cors` is used to enable cross-origin resource sharing, allowing other origins to access the API endpoints, which is particularly useful when testing. By using this as our first middleware, we enable CORS on app requests to `api`.

```
const functions = require('firebase-functions');
const express = require('express');
const mysql = require('mysql');
const cors = require('cors');
const api = express();
api.use(cors({ origin: true }));
```

10.2 mysql.createConnection()

The `con` variable stores the MySQL connection, which is created using the host, credentials, and database name for the database in question.

```
var con = mysql.createConnection({
    host: "localhost",
    user: "user",
    password: "pass",
    database: "mydb"
});
```

10.3 con.connect()

Attempts to use the credentials stored in `con` to connect to the database. Creates a log message if successful, throws an error otherwise.

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

10.4 readDataFromMySQL()

Reads the entire `data` table from the MySQL database and sends the result to the callback function.

```
function readDataFromMySQL(cb) {
```

```
    con.query("SELECT * FROM data;", function (err, data, fields) {
        if (err) throw err;
        cb(data);
    });
}
```

10.5 readTypesFromMySQL()

Reads all types stored in the `types` table from the MySQL database and sends the result to the callback function.

```
function readTypesFromMySQL(cb) {
    con.query("SELECT * FROM types;", function (err, types, fields) {
        if (err) throw err;
        cb(types);
    });
}
```

10.6 writeToMySQL()

Clears the existing `data` table using `TRUNCATE TABLE data`.

Inserts a new row for each row in the HTML table, containing all the necessary information.

Clears the existing `types` table using `TRUNCATE TABLE types`.

Inserts each type in the web app as a new row.

```
function writeToMySQL(jsonData) {
    var data = jsonData[0];
    var types = jsonData[1];

    con.query("TRUNCATE TABLE data;");

    for(var i = 1; i < data.length; i++) {
        con.query("INSERT INTO data (id, date, account, type, security,
            ↪ amount, dAmount, costBasis, files) VALUES ('" + data[i].join
            ↪ ("','") + "');");
    }

    con.query("TRUNCATE TABLE types;");

    for(var i = 0; i < types.length; i++) {
        con.query("INSERT INTO types (typename) VALUES ('" + types[i] +
            ↪ "');");
    }
}
```

10.7 api.post()

Handles `POST` requests, which are used to update the database.

Passes the body of the request to the `writeToMySQL()` function. Sends the body back with a success code.

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

10.8 api.get()

Handles `GET` requests, which are used to retrieve data from the database.

Calls the `readDataFromMySQL()` and `readTypesFromMySQL()` functions, sends the return values back in an array, along with a success error code.

```
api.get('/api', (req, res) => {
    readDataFromMySQL(function(data) {
        console.log(data);
        readTypesFromMySQL(function(types) {
            console.log(types);
            return res.status(200).send([ data, types ]);
        });
    });
});
```

10.9 exports.api

Requests which are routed by firebase to the endpoint are handled by the `api` object.

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

11 CSS

11.1 Vertical Scrolling Table

The table id refers to the `article` that contains the table, rather than the table itself. This article was given a max height of 80 visual heights, or approximately 80% of the screen height. `overflow: auto;` specifies that, if necessary, a scroll bar should be present, this is true for both horizontal and vertical scrolling.

```
#table {
    max-height: 80vh;
    overflow: auto;
}
```

The last three properties below are important for keeping the header in place while scrolling.

```
th {
    min-width: 200px;
    width: 10%;
    position: sticky;
    background: white;
    top: 0;
}
```

- `position: sticky;` is used to keep the object in place when scrolling.
- `background: white;` is used to give the element a non-transparent background, so that data cannot be seen through the header.
- `top: 0;` is used to specify that the element should remain at the top of its parent element with no offset.

11.2 Horizontal Scrolling on Overflow

The inputFields id is used to identify the `article` element that contains the `form`. The important property here is the `overflow-x: auto;` line, which specifies that, if the child element is wider than this element, a horizontal scroll bar should be present.

The `form` element, which is a child of the `article` is given a minimum width to ensure that the scroll bar is created rather than reducing the width of the element.

```
#inputFields {
    padding: 10px 0;
    overflow-x: auto;
}

form {
    min-width: 1900px;
}
```

11.3 Miscellaneous

11.3.1 Sort buttons

The header cells contain two `sections`, one of which has a special class. Both sections are given a `margin` and `padding` of 0 to minimise wasted space. Both sections are also set to `display: inline-block` to specify that they should be arranged horizontally.

Both sections are given a `width` of 80% of the parent element. However, this is overruled for the element with a class of `sort`, which is assigned a `width` of 10% to ensure that both elements fit horizontally in the parent.

To further reduce wasted space, the `border` and `padding` of the buttons are set to 0. The button's `display` ↪ property is set to `block` as otherwise, it would inherit the `inline-block` property from its parent and attempt to arrange horizontally. Lastly, the buttons are set to take up the entire width of the parent element.

```
th > section {
    width: 80%;
    display: inline-block;
    padding: 0;
    margin: 0;
}

.sort {
    width: 10%;
}

.sort > button {
    padding: 0;
    border: 0;
    display: block;
    width: 100%;
}
```

11.3.2 Editing highlight

The current row selected for highlighting is specified using a class. As such, it is possible to give this row unique styling. For example, currently the row is given a yellow background.

```
.editing {
    background-color: yellow;
}
```

11.3.3 Table borders

Table borders do not render properly with a scrolling body and fixed header. To resolve this, borders are rendered using the `box-shadow` property.

Two box shadows are defined, one extends outwards from the right and bottom by one pixel in each direction. The other is given the `inset` property, so that it extends inwards from the left and top. This creates a full

border with a width of 1 pixel in each direction. As these borders are offset from the element, adjacent elements will overlap borders, preventing borders from combining into extra thick borders.

```
#table,
table,
td,
th {
    box-shadow: 1px 1px black, inset 1px 1px black;
}
```