Criterion C: Development

Summary of Techniques:

- Google Sheets Update
  - Edit Liveschool Points on Google Sheets using Google Sheets API
- Login Authentication
  - Check for name and ID match between Google Sheets and user input using Fetch API
- Data Display
  - Display customer data associated with their ID upon customer authentication using Fetch API and localStorage to pass data between pages
  - Display customer data associated with ID once admin inputs ID using Fetch API and localStorage
- Import and Synthesize Data using Google Sheets

Note: Fetch API sends a request and gets response from Google Sheets
Note: CSS not included as major technique (Vidas, "CSS Tutorial", et al.)
Note: Logout not included as major technique; redirects to Login.html

Google Sheets Update:

Edit Liveschool Points:

To edit Liveschool points on my "Data Mastersheet" sheet on my Google Sheets, I used Google Sheets API (Google Developers). I tried Fetch API, but while I could figure out how to read the spreadsheet with it, I could not figure out editing (See Crit B: Record of Tasks). After an error-filled general setup and authentication, I created objects for the ID and Liveschool points columns in my "Data Mastersheet" and "Data From Today" sheets (Learn Google Spreadsheets). To actually grab the values from the columns, I used Google Sheets Get API.

Figure 1: Part of Enabling API (console.developers.google.com)

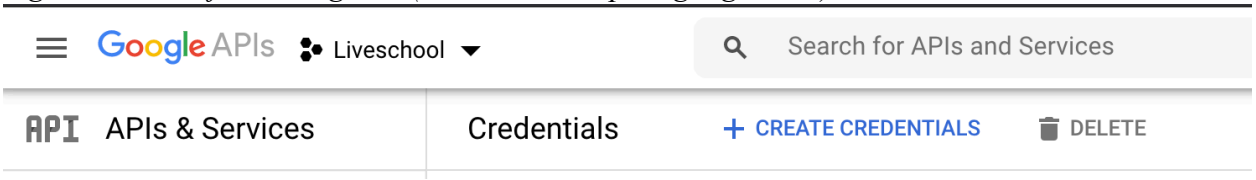


Figure 2: Creating Objects to Retrieve Data

```
//IDs from Data From Today
const idsPoints = {
    spreadsheetId: '1dZ0nXkkSDKRbFak1osm4Ekq3F2VdX-xPMmlI6Do7J1w',
    range: 'Data From Today!A2:A100'

};
//IDs from Data Mastersheet
const idsTotalPoints = {
    spreadsheetId: '1dZ0nXkkSDKRbFak1osm4Ekq3F2VdX-xPMmlI6Do7J1w',
    range: 'Data Mastersheet!A2:A100',
};
//points from Data From Today
const points = {
    spreadsheetId: '1dZ0nXkkSDKRbFak1osm4Ekq3F2VdX-xPMmlI6Do7J1w',
    range: 'Data From Today!C2:C100'
};
//points from Data Mastersheet
const totalPoints = {
    spreadsheetId: '1dZ0nXkkSDKRbFak1osm4Ekq3F2VdX-xPMmlI6Do7J1w',
    range: 'Data Mastersheet!C2:C100'
};
```

Figure 3: Creating Arrays from Data Columns

```
let totalPointsValue = await gsapi.spreadsheets.values.get(totalPoints);
let totalPointsArray = totalPointsValue.data.values;
```

Figure 4: parseInt() to compare arrays later

```
for (i = 0; i < pointsArray.length; i++) {
    pointsArray[i] = parseInt(pointsArray[i], 10);
}
```

I used nested loops to **search** through my IDs columns as nested loops would allow me to check if the ID from “Data From Today” matched an ID from “Data Mastersheet” after iterating fully through the latter each time. I also used procedural decomposition, as when I found a match, I referred to my function updateLiveschoolPoints, which essentially created an object for the cell of “Data Mastersheet” which added the two Liveschool points values together. This updated the “Data Mastersheet” completely for display.

Figure 5: Finding a Match

```
    for (i = 0; i < pointsIdArray.length; i++) {
        for (j = 0; j < totalPointsIdArray.length; j++) {
            if ((parseInt(pointsIdArray[i], 10) == parseInt(totalPointsIdArray[j], 10))) {
                range = "Data Mastersheet!C" + (j + 2);
                totalPointsArray[j] += pointsArray[i];
                updateLiveschoolPoints(range, totalPointsArray[j], gsapi);
            }
        }
    }
```

Figure 6: Updating Cell with New Points

```
function updateLiveschoolPoints(range, dataToUpdate, gsapi) {
    const updater = {
        spreadsheetId: '1dZ0nXkkSDKRbFak1osm4Ekq3F2VdX-xPMmlI6Do7J1w',
        range: range,
        valueInputOption: 'USER_ENTERED',
        resource: { values: [[dataToUpdate]] }
    };

    let res = gsapi.spreadsheets.values.update(updater);

}
```

Figure 7: Sample of "Data From Today" sheet

1234567	Sam Robertson	7
5239014	Nilla Maferson	5
9538492	Michael Minkcat	4
3981203	Emily Ramends	7
3712034	Schmidt Setson	6
4213943	Cathy Wool	7
5832910	Joe	7
3638401	Robert Bobbers	12
3123479	Petersen Dixon	8

Figure 8: Before and After of "Data Mastersheet"'s Joe

1293042	Brandon Jones	100
5832910	Joe	233
1239583	Jones Jill-Jones	100

1293042	Brandon Jones	100
5832910	Joe	240
1239583	Jones Jill-Jones	100

Login Authentication

For login, I opted to go with an ID-Name approach, preferred by my client due to its simplicity. We chose 7-digit IDs, as the probability of guessing the right 7-digit ID for a person is astronomically low, increasing security (See Appendix B). I also stored the input type as password in HTML so the ID wasn't visible during entering (Costa). Hence, for login, I used Fetch API to read data from the "Data Mastersheet." I opted to go with Fetch API as Google Sheets API was proving especially difficult to integrate with HTML and CSS (See Crit B: Record of Tasks), even though it worked perfectly with the spreadsheet itself ("JavaScript Fetch API Explained by Examples"). I created an object to store my HTML references for ease of use in my JavaScript to utilize user input ("JavaScript Tutorial").

Figure 9: Login.html to Login.js to UI Connections

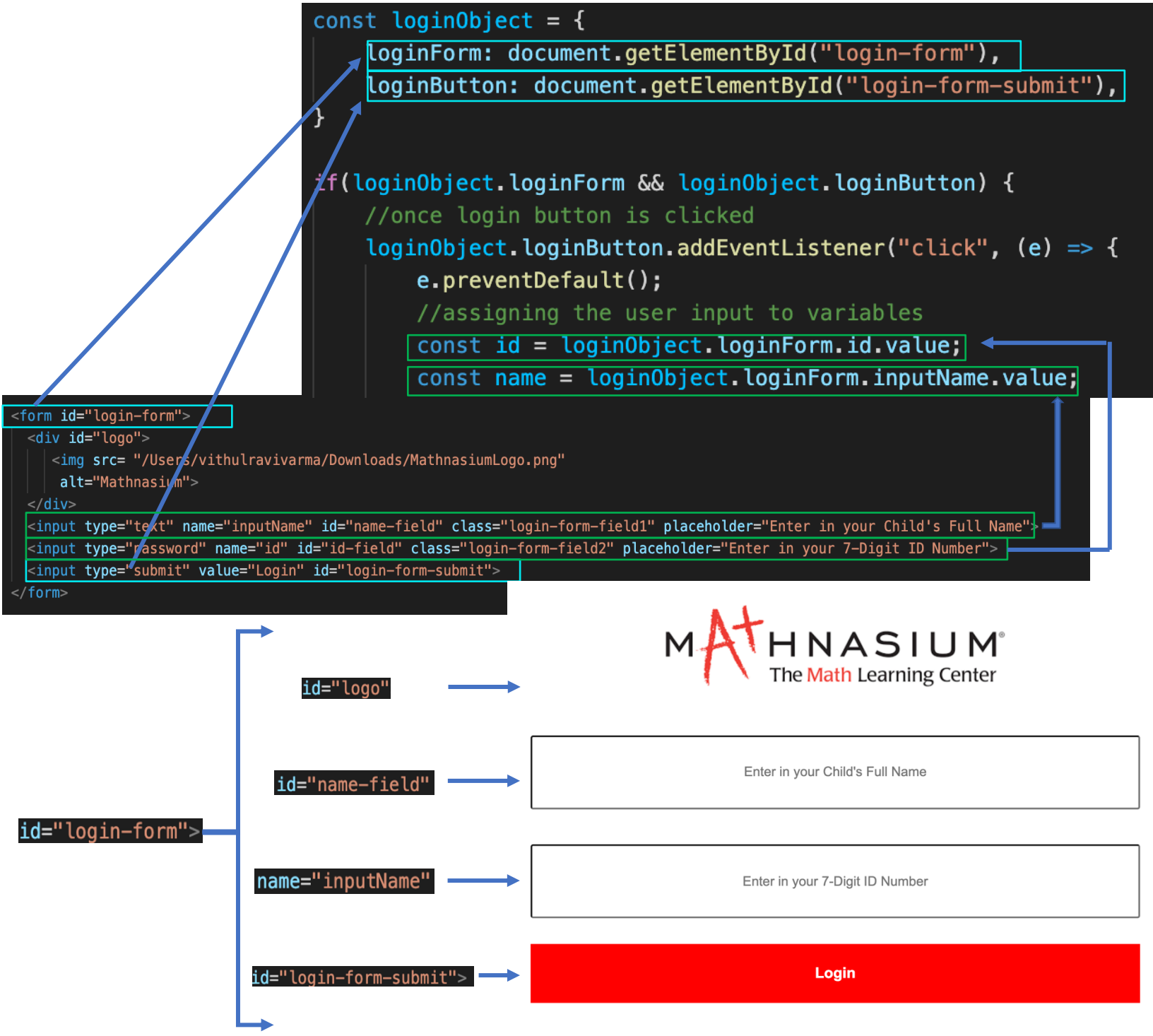


Figure 10: Fetch API reads and gets data

```
fetch('https://sheetdb.io/api/v1/9z8nttlgn3em0')
    .then(function (response) {
        response.json()
            .then(function (dataObject) {
```

Then, I used a for loop in my function loginAndGetData to iterate through the array of data from the spreadsheet, row by row, to check for a name-ID match.

Figure 11: loginAndGetData Admin and Database Checking

```
function loginAndGetData(dataObject, id, name) {
    for (i = 0; i < dataObject.length; i++) {
        if (id == dataObject[i].Id) {
            idFound = true;
        }
        if (name.toLowerCase() == dataObject[i].Name.toLowerCase()) {
            nameFound = true;
        }
        if (id == "0123456" && name.toLowerCase() == "Admin".toLowerCase()) {
            isLogged = true;
            locNew = loc.replace("Login", "Admin");
            location.replace(locNew);
        }
```

The for loop helped me distinguish name-ID matches on a specific row, helping me escape a bug of valid credentials but the pair not being on the same row. When they matched specifically, I used my own function, localStorage, to store my data due to its simplicity and its functionality in passing values between JavaScript files (“HTML Tutorial”). It also helped me grab and store the specific values on the row in conjunction with dataObject[i].

Figure 12: Customer Authentication

```
//if the user enters correct credentials
if (id === dataObject[i].Id && name.toLowerCase() === dataObject[i].Name.toLowerCase()) {


    //user is now logged in
    isLogged = true;

    //setting the values from the row in local storage to access in the next page
    storeData(dataObject);
    //redirecting the user to their student's data
    locNew = loc.replace("Login", "Customer");
    location.replace(locNew);
}
```

Figure 13: Benefit of For Loops and Arrays

1923412	Jake Harp
9538492	Michael Minkcat

dataObject[i] ensures that user input of “1923412” and “Michael Minkcat” does not yield a successful login

For error validation, my key algorithmic thinking was displayed through my use of ASCII to ascertain whether the user input was fully numerical. I also utilized my nameFound and idFound booleans to create the name and ID don’t match error.

Figure 14: function errorValidation

```
function errorValidation(id, idFound, nameFound) {
    let isItAllInt;
    //converting to values to ASCII
    for (i = 0; i < id.length; i++) {
        let intVersion = id.charCodeAt(i);
        //checking if it is an actual number through ASCII
        if (intVersion < 58 && intVersion >= 48) {
            isItAllInt = true;
        } else {
            isItAllInt = false;
            i = id.length;
        }
    }
    //checking length and if it's fully numeric
    if (id.length != 7 || isItAllInt == false) {
        alert("Please enter a 7-Digit ID: only numbers");
    //if a name or id was found, but they were not logged in, something was
    //wrong with the input
    } else if (nameFound == true || idFound == true) {
        alert("The name and the ID do not match");
    } else {
        alert("Please enter a valid name and ID");
    }
}
```

```
if (id == dataObject[i].Id) {
    idFound = true;
}
if (name.toLowerCase() == dataObject[i].Name.toLowerCase()) {
    nameFound = true;
}
```

Data Display

Customer Data:

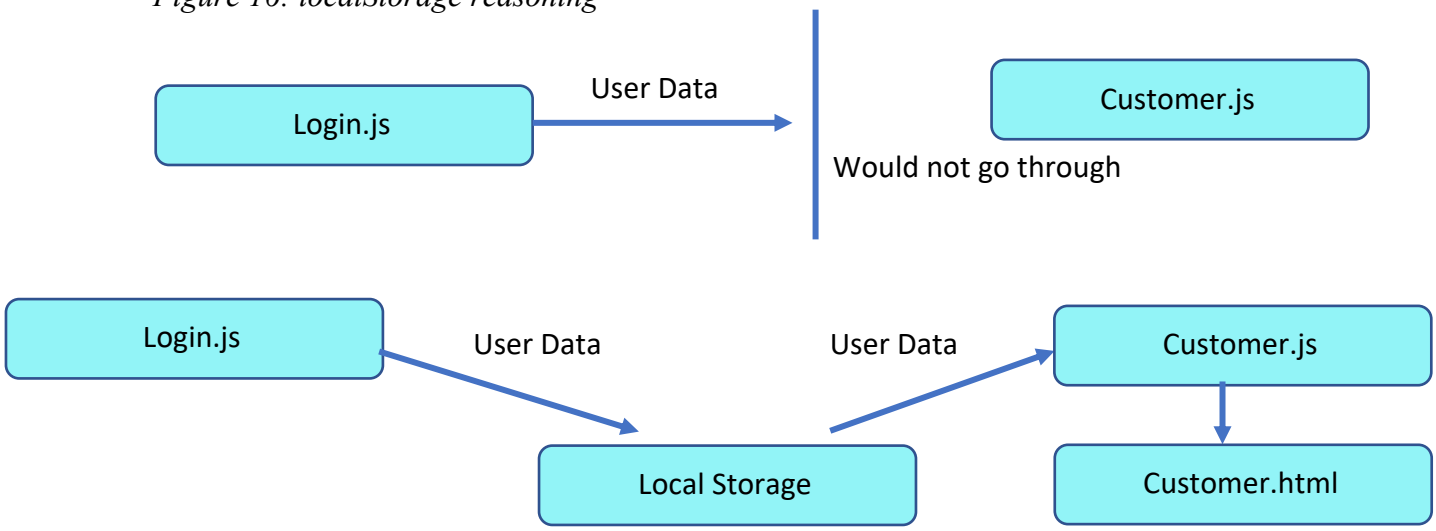
To display customer data, I got data from localStorage, returned it from a function, and used it in another to link values to their respective HTML elements, the only method that I used which worked in transferring data throughout HTML.

Figure 15: Local Storage Data Display

```
function getDataFromLocalStorage() {
    let studentData = {
        liveschoolPoints: localStorage.getItem("liveschool"),
        theName: localStorage.getItem("name"),
        comment1: localStorage.getItem("comment1"),
        comment2: localStorage.getItem("comment2"),
        comment3: localStorage.getItem("comment3"),
        comment4: localStorage.getItem("comment4"),
        comment5: localStorage.getItem("comment5")
    };
    return studentData;
}
```

```
function setDataInHTML(studentData) {
    document.getElementById("liveschool").innerHTML = studentData.liveschoolPoints;
    document.getElementById("name").innerHTML = "Hi " + studentData.theName + "!";
    document.getElementById("comment1").innerHTML = "Comment 1: " + studentData.comment1;
    document.getElementById("comment2").innerHTML = "Comment 2: " + studentData.comment2;
    document.getElementById("comment3").innerHTML = "Comment 3: " + studentData.comment3;
    document.getElementById("comment4").innerHTML = "Comment 4: " + studentData.comment4;
    document.getElementById("comment5").innerHTML = "Comment 5: " + studentData.comment5;
}
```

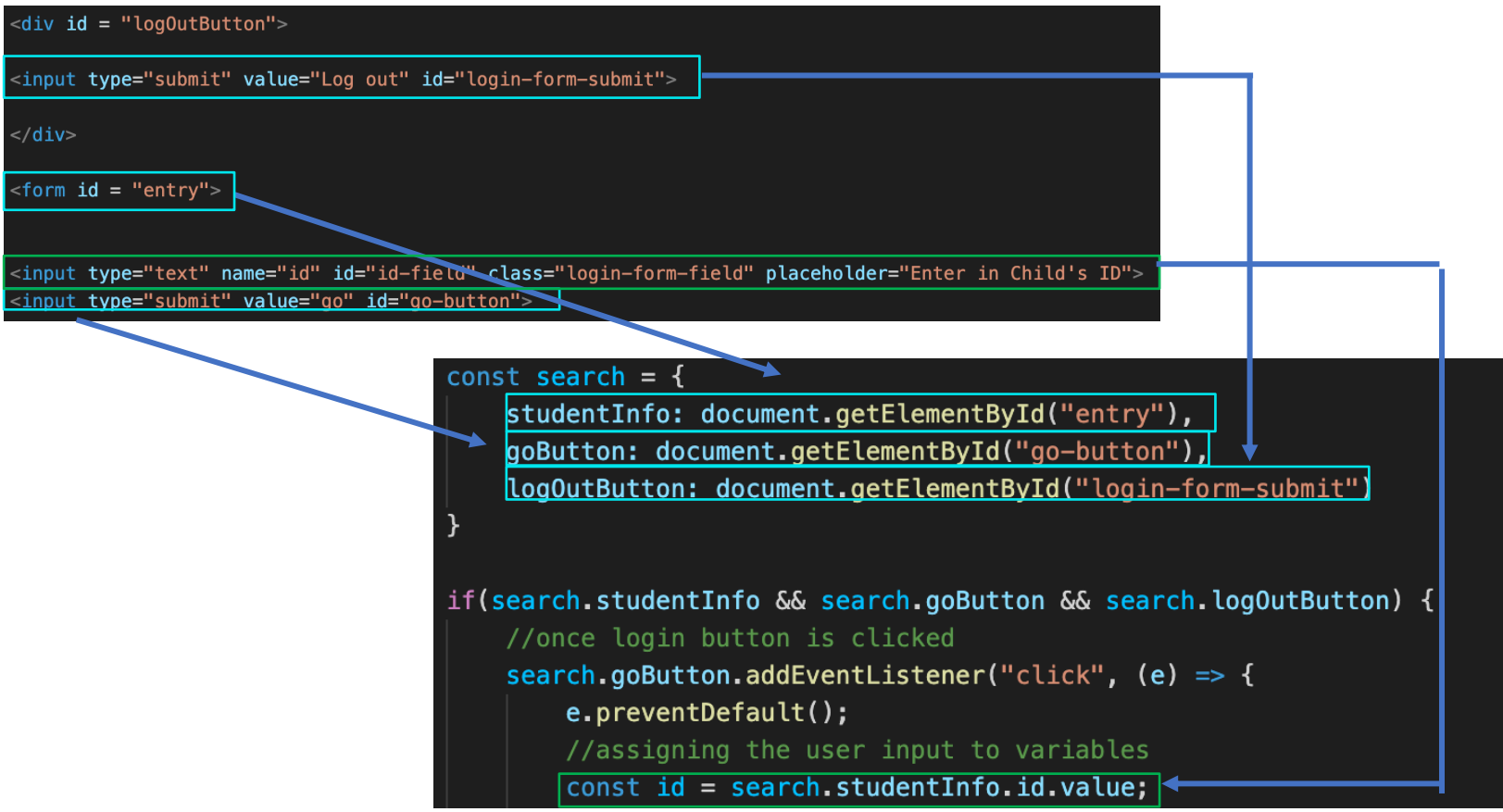
Figure 16: localStorage reasoning



Admin Data:

Initially, I had blank boxes without the customer data show for the whole time, but upon the advice of my advisor, I used the display and block functionality of JavaScript to toggle what parts to show (Uniyal). Functionality-wise, I reused many parts of my code for Login.js, with slight modifications. I used a search object with references to HTML for organization and easy functionality.

Figure 17: Admin.js connection with Admin.html



I used Fetch API to read data from the "Data Mastersheet" as I compared the user input from the Search object with the database. I used procedural decomposition to create a searchAndDisplay function that this is all stored in. The program **searches** the spreadsheet for the id, and if it matches, it stores the data in localStorage and passes to HTML elements using my custom-built function storeData. After this, the data shows. I used arrays and for loops, the most efficient way to iterate through spreadsheet columns.

Figure 18: Credentials For Loop

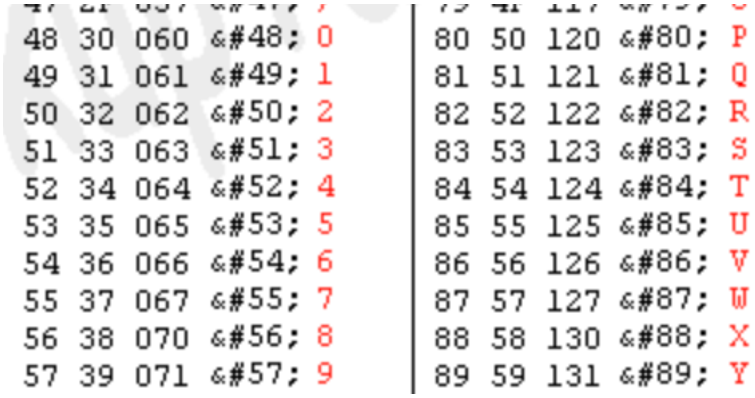
```
for (i = 0; i < dataObject.length; i++) {

    //if the user enters correct credentials
    if (id === dataObject[i].Id) {
        isFound = true;
        //store data in HTML IDs
        storeData(dataObject);
        //showing text
        document.getElementById("mainHolder").style.display = "block";
    }
}
```

Like before, I used error validation (see Test Plan from **Criterion B** for more details on all error alerts), reusing my earlier to check for non-numbers using ASCII: numbers have ASCII values between 48 and 57 ("Additional Reference Material: The Standard ASCII Character Set and Codes"). My procedural decomposition in breaking up Admin.js into multiple functions greatly aided my debugging and accessibility of my code.

Figure 19: Error Validation

```
function errorValidation(id) {
    let isItAllInt;
    //converting to values to ASCII
    for (i = 0; i < id.length; i++) {
        let intVersion = id.charCodeAt(i);
        //checking if it is an actual number through ASCII
        if (intVersion < 58 && intVersion >= 48) {
            isItAllInt = true;
        } else {
            isItAllInt = false;
            i = id.length;
        }
    }
    //checking length and if it's fully numeric
    if (id.length != 7 || isItAllInt == false) {
        alert("Please enter a 7-Digit ID: only numbers");
    //if a name or id was found, but they were not logged in, something was
    //wrong with the input
    } else {
        alert("ID not found in database");
    }
}
```

Figure 20: why ASCII?



48	30	060	&#48;	0	80	50	120	&#80;	P
49	31	061	&#49;	1	81	51	121	&#81;	Q
50	32	062	&#50;	2	82	52	122	&#82;	R
51	33	063	&#51;	3	83	53	123	&#83;	S
52	34	064	&#52;	4	84	54	124	&#84;	T
53	35	065	&#53;	5	85	55	125	&#85;	U
54	36	066	&#54;	6	86	56	126	&#86;	V
55	37	067	&#55;	7	87	57	127	&#87;	W
56	38	070	&#56;	8	88	58	130	&#88;	X
57	39	071	&#57;	9	89	59	131	&#89;	Y

Import and Synthesize Data using Google Sheets:

See Criterion B, Algorithms and Appendix, Transcripts of Conversations for detail.

Figure 21: Sample Query

```
=QUERY(DaySplitLBSC!A3:AH, "Select A where F contains '1'
```

Figure 22: Converting alphabet to numbers

```
=LEN(D2)- LEN(SUBSTITUTE(lower(D2),"y",""))
```

Word Count: 1037