Group Members: Timothy Dakis

# Graypaper

## Code Snippet(s) #1

```
function welcomeTextAnimation() {
    // https://github.com/bradtraversy/50projects50days/blob/master/auto-text-effect/script.js
    const textEl = document.getElementById('welcome-text')
    const text = 'Welcome back, <a href="profile-page/profile.html">Timothy</a>!'
    let textIdx = 1

    // determine where all the < and > of the opening and closing anchor tags are
    let AnchorOpenStartIdx = text.indexOf('<');
    let AnchorOpenEndIdx = text.indexOf('>');
    let AnchorCloseStartIdx = text.lastIndexOf('<');
    let AnchorCloseEndIdx = text.lastIndexOf('>');
    let speed = 200;

    writeText()

    function writeText() {
        textEl.innerHTML = (() => {
            // check if index of text has reached on of the opening <, if so set it to the index of the closing >
            textIdx = textIdx === AnchorOpenStartIdx ? AnchorOpenEndIdx + 1 :
                      textIdx === AnchorCloseStartIdx ? AnchorCloseEndIdx + 1 :
                      textIdx;
            return text.slice(0, textIdx);
        })();

        textIdx++;

        if(textIdx <= text.length) {
            setTimeout(writeText, speed);
        }
    }
}

welcomeTextAnimation();
```

### Explanation

This function essentially iterates through a string character by character to animate the text on the web page such that it builds up the sentence letter by letter. It does this by setting a timeout to call the writeText function, where each time writeText is called an index is increased and that index is used to calculate the end point of the substring we are taking to render on the webpage where eventually the entire string will be displayed.

One big issue I had with this was trying to figure out how to deal with the anchor <a> </a> tags in the string, as it did not look nice when it would partially type out the tag, then hide the tag when it was closed. To fix this issue I simply found where the anchor tags opened and closed, then checked if the index had reached a point in the string where the “<” of a tag was reached, and if it did I incremented the index enough such that it fully covered the anchor tag and also added in the next character so that it was still a smooth animation on the webpage.

## Code Snippet(s) #2

```
// https://stackoverflow.com/questions/4295782/how-to-process-post-data-in-node-js
app.put('/data/add-user-data', (req, res) => {
    let dataFileName = `data/user-data.json`;
    fs.readFile(dataFileName, (err, data) => {
        if(err) throw err;
        let userData = JSON.parse(data);
        // if there is no data in JSON, create a skeleton to insert data
        if(!userData['data-exists']){
            userData['data-exists'] = true;
            userData['num-playing'] = 0;
            userData[`num-completed`] = 0;
            userData['num-planning'] = 0;
            userData.games = {};
        }
        // if game already in profile, decrease counter tracking its current status and just update its status in the JSON to new status
        if(Object.hasOwn(userData.games, req.body.data['game-id'])) {
            userData[`num-${userData.games[req.body.data['game-id']].status}`] -= 1;
            userData.games[req.body.data['game-id']].status = req.body.data.status;
        }
        // otherwise, create a new entry with all relevant info within the JSON
        else {
            userData.games[req.body.data['game-id']] = {
                'img-url': `../game-cover-images/covers-200x300/${req.body.data['img-file-name']}`,
                'name': req.body.data.name,
                'rating': "Not Rated",
                'status': req.body.data.status
            };
        }
        // increase counter tracking the new status of the game
        userData[`num-${req.body.data.status}`] += 1;
        fs.writeFile(dataFileName, JSON.stringify(userData, null, 4), (err) => {if(err) throw err;});
        // https://stackoverflow.com/questions/9107226/how-to-end-an-express-js-node-post-response
        res.status(204).send();
    })
})
```

### Explanation

When the user tries to add something to or update something in their profile, a PUT request is made and this snippet is called in the server. What this snippet does essentially is it loads in the data of the user stored in the server, and then it checks if any data is currently stored in the JSON. If there is no data in the JSON, then it creates a skeletal structure in the JSON for the sent in data to be stored into.

Afterwards, a check is done to see if the user is simply trying to update a game already listed in their profile, this is done by checking if the game is currently in the JSON by using the id associated with the game. If the game exists in the profile so that it is simply an update, all that is done is decreasing the counter tracking the current status of the game, and then changing the status of the game in the profile. Otherwise, if a game is being added, an object is constructed with relevant data and stored at a property associated with the game's id. In either case, a counter tracking what the new status will be is increased.

One challenge I faced was how to properly access the properties of the JSON stored in the server using the data sent in with the PUT request by the user. E.g. like this line:

```
userData[`num-${userData.games[req.body.data['game-id']].status}`] -= 1;
```

The solution was simply to understand what the structure of the data in both the JSON in the server and the JSON sent with the request looked like, such that I could construct a string using a template literal to access the property in the server-side JSON.

For example, the above line constructs the property name of the correct counter to decrease, and then decrease it. It does that by pre-pending "num-" to the value returned by accessing the "status" property of the correct object on the server-side JSON using the game-id sent by the user in the request body.

# Code Snippet(s) #3

```
<div class="rating-container hide">
    <label>
        Rating
        <input type="text" class="rating-field" data-game-id="${gameListData.games[i].id}" placeholder=".../5">
    </label>
    <span class="error-text"></span>
```

```
function addRatingBoxListeners(){
    const ratingBoxes = document.querySelectorAll('.rating-field');
    ratingBoxes.forEach(ratingBox => {
        ratingBox.addEventListener('keydown', (e) => {
            // clear any error text on input
            let errorPrompt = e.target.parentNode.nextElementSibling;
            errorPrompt.innerText = '';
            if(e.key === 'Enter'){
                // check if input is not a number between 0-5, and output an error if that is the case
                if(!e.target.value.match(/^\d+(.\d+)?$/) || parseFloat(e.target.value) < 0 || parseFloat(e.target.value) > 5) {
                    errorPrompt.innerText = 'Input a number beween 0-5';
                }
                else {
                    // make PUT request to update JSON in the server
                    fetch('/data/add-user-review', {
                        method: 'PUT',
                        headers: {
                            'Content-Type': 'application/json'
                        },
                        body: JSON.stringify({
                            data: {
                                    'game-id' : e.target.dataset.gameId,
                                    'review' : e.target.value
                            }
                        })
                    });
                    // clear rating input box on successful submisson
                    e.target.value = '';
                }
            }
        })
    })
}
```

## Explanation

The first snippet is just a few lines of another function used to build the HTML in the webpage. It was added here to showcase the lack of form with the input tag (which is relevant for explaining the issue I had faced and needed to resolve).

The second snippet is the main focus. So what it is doing is for every rating box, it adds an event listener that checks for any key presses. When any key is pressed, any shown errors disappear. However, when “enter” specifically is pressed, a test for validity is done to check if the user’s input is indeed a number between 0 and 5. If it is not, then text is added to the error prompt, and this gets displayed to the user. Otherwise, if it is a number between 0 and 5, then the Fetch API is used to make a PUT request sending the game id and the rating value to the server so the server can update the relevant portion of the user’s profile data JSON. After it is sent, the input field is cleared.

One issue I faced was how to prevent the webpage refreshing when the data was sent. To resolve the issue I did a fair bit of research and came to the conclusion that the issue stemmed from the fact I had initially tried using a form to send the data. A form is limited to GET and POST methods, where GET obviously does not make sense in this context, but POST could; however, the use of POST was what was causing the page to refresh. So, to avoid this issue, I simply removed the form and added my own validity checks to the data being entered, and then sent the data to the server using a PUT request which does not refresh the web page, and was technically the more appropriate method to use given a PUT request should be used for updating data.

# Code Snippet(s) #4

```
function parseData(data) {
    // if data exists, then parse the data
    if(data['data-exists']) {
        // add relevant data to user stats counters, then start the visual incrementing
        document.getElementById('playing-data').dataset.target = data['num-playing'];
        document.getElementById('complete-data').dataset.target = data['num-completed'];
        document.getElementById('planning-data').dataset.target = data['num-planning'];
        beginIncrementingCounters();

        // get relevant DOM elements
        const PlayingTable = document.getElementById('playing-table');
        const CompletedTable = document.getElementById('completed-table');
        const PlanningTable = document.getElementById('planning-table');

        // add each game to the relevant table
        const gameData = data.games;
        for(const gameId in gameData) {
            let htmlToAppend =
                `
                <tr>
                    <td><img src="${gameData[gameId]['img-url']}" alt="${gameData[gameId].name} cover art"></td>
                    <td>${gameData[gameId]['name']}</td>
                    <td>${gameData[gameId]['rating']}</td>
                </tr>
                `
            // pick which table to append data to
            let tableToAppendTo = gameData[gameId]['status'] === 'playing' ? PlayingTable :
                gameData[gameId]['status'] === 'completed' ? CompletedTable : PlanningTable;
            tableToAppendTo.insertAdjacentHTML('afterbegin', htmlToAppend);
        }
    }
    // if no data exists, add a prompt leading them to the game list so they can start tracking
    else {
        let htmlToAppend =
        `
        <div class="no-data-prompt">
            It looks like your profile is empty<br>
            Check out the <a href="../game-list-page/game-list.html">Game List</a> to add some games!
        </div>
        `
        document.querySelector('.main-page-content').innerHTML = htmlToAppend;
    }
}
```

## Explanation

So this function is used to parse the data received by the browser from the server. This function specifically is related to the feed page, though there are similar functions to parse the data in the game list and user profile pages.

What this function basically does is it checks if the user's data exists. If it does, it first accesses several elements used for counters and sets their target to the corresponding counters stored in the user's data. Afterwards, the function constructs variables to store the various tables we may be adding data to. Then it loops through all the objects stored in the user's data, where each object represents a different game stored in the user profile, and builds up some HTML to append. The function properly builds up the HTML by using a template literal and accessing the object's property where relevant to return a string to be placed in that part of the HTML. Then, the function selects the correct table to add all this HTML to by checking the status property of the current object being looked at.

Though, if no data is found, then a prompt is simply inserted redirecting the user to the game list page so that the user can begin tracking things to populate their profile page

There were no real challenges or issues of note faced as it was fairly straightforward to implement. Though, concerned about efficiency to a degree, I researched what the best way to keep adding HTML to a DOM element was, and that resulted in me using the "insertAdjacentHTML()" function as opposed to simply doing "element.innerHTML +=" due to inherent differences in how both work even if the outcome is the same.

# Code Snippet(s) #5

## Explanation

```
function addButtonListeners(){
    function addPlayingButtonListeners(){
        const playingButtons = document.querySelectorAll('.playing-button');
        playingButtons.forEach(playingButton => {
            playingButton.addEventListener('click', (e) => {
                const siblingsList = e.target.parentNode.children;
                let ownChildIdx = 0;
                // hide its own button and show the other buttons and the rating box
                addOrHideButton(siblingsList, ownChildIdx)
                hideElement(e.target.parentNode.nextElementSibling, false);
                sendButtonData(e.target);
            })
        })
    }
    function addCompletedButtonListeners(){
        const completedButtons = document.querySelectorAll('.completed-button');
        completedButtons.forEach(completedButton => {
            completedButton.addEventListener('click', (e) => {
                const siblingsList = e.target.parentNode.children;
                let ownChildIdx = 1;
                // hide its own button and show the other buttons and the rating box
                addOrHideButton(siblingsList, ownChildIdx);
                hideElement(e.target.parentNode.nextElementSibling, false);
                sendButtonData(e.target);
            })
        })
    }
```

```
function addPlanningButtonListeners(){
    const planningButtons = document.querySelectorAll(`.planning-button`);
    planningButtons.forEach(planningButton => {
        planningButton.addEventListener('click', (e) => {
            const siblingsList = e.target.parentNode.children;
            let ownChildIdx = 2;
            // hide its own button and the rating box and show the other buttons
            addOrHideButton(siblingsList, ownChildIdx);
            hideElement(e.target.parentNode.nextElementSibling);
            sendButtonData(e.target);
        })
    })
}
function addRemoveButtonListeners(){
    const removeButtons = document.querySelectorAll('.remove-button');
    removeButtons.forEach(removeButton => {
        removeButton.addEventListener('click', (e) => {
            let siblingsList = e.target.parentNode.children;
            let ownChildIdx = 3;
            // hide its own button and the rating box and show the other buttons
            addOrHideButton(siblingsList, ownChildIdx)
            hideElement(e.target.parentNode.nextElementSibling);
            // make PUT request to update JSON in the server
            fetch('/data/remove-user-data', {
                method: 'PUT',
                headers: {
                    'Content-Type': 'application/json'
                },
                body: JSON.stringify({
                    data: {
                            'game-id' : e.target.dataset.gameId
                    }
                })
            });
        })
    })
}
```

One of the issues I faced early on was how to hide the button the user clicked, and ensure the other buttons all show as well, and then hide or show the rating box when appropriate (e.g., if the user is planning to play a game, it does not make sense to prompt them for a review given they have not played it yet).

So to resolve it, 4 types of buttons were made, and each type of button has an event listener added to it that checks when it is clicked. Each type of button, when clicked, first gets a list containing itself and all its siblings (i.e. if the user clicks a “completed” button, it gets itself and the other 3 buttons associated with it). After that, a value is hardcoded so that the button knows what its index is within the list (as all buttons are placed in a fixed order within the HTML). Next, a function is called which will hide itself and make all other buttons appear (if any were hidden). Then, if a “completed” or “playing” button is clicked, it ensures the rating box appears, otherwise it will disappear.

Then, if any button except the “remove” button is pressed, a function is called to send data to the server. Otherwise, if the remove button is clicked, the Fetch API is used to make a PUT request to the server to send the game id so that the server can update the user’s JSON of data by removing the record of the game associated with the sent in game id.