NM2207 Individual Reflection

**Ideation**

I began this project by wondering how my financial decisions and emotional experiences might inform one another when recorded side by side, also with the intention of making use of what was covered in the course in an extensive way. I sketched a simple two‑column interface on paper in which the left column would collect income and expense entries and the right column would display a monthly calendar for mood tagging.

Translating those paper mock-ups into static HTML (feels like laying bricks or preparing materials to be used for construction) allowed me to confirm that elements such as <button id="ExpenseBtn"> and <div id="calendarGrid"> would sit comfortably on the page before adding any JS scripting for interactivity.

**A screenshot of a mood tracker

Description automatically generated**

*Overall UI Planned*

*A screenshot of a calendar

Description automatically generated*

*Actual Final UI Layout*

***A close-up of a form

Description automatically generated*A menu with text and numbers

Description automatically generated with medium confidence**

*Budget Tracker: Planned Interactions and Pop-ups*

**A screenshot of a calendar

Description automatically generated**

*Mood Tracker: Planned Interactions and Pop-ups*

**Planning and Structuring**

After finalising the layout, I began by outlining the core data models in pseudocode comments at the top of scripts/script.js. I created an object called categoryData that associates each category, represented as an uppercase string, with an object containing three properties: a type (indicating whether it's an expense or income), a colour (which stores the category’s colour), and an array of records that track transactions within that category.

At the same time, I defined a moodData object, which is keyed by date strings. Each date holds one of four possible mood values, capturing the user's mood for that day.

In addition to the data models, I planned the flow of operations in the code to ensure everything ran smoothly. I structured the startup sequence to follow a "load, render, bind" pattern.

Initially, the code calls loadCategories() and loadMoods() to load the data. Then, it runs renderCategories() and generateCalendar() to display the categories and calendar. Finally, event listeners are attached to interactive elements, such as the expenseBtn, incomeBtn, and each cell in calendarGrid.

**Implementation of the Budget Module**

To capture transactions, I created two top-level buttons: “+ EXPENSE” and “+ INCOME”. These buttons are accessed using document.getElementById("ExpenseBtn") and document.getElementById("IncomeBtn").

Clicking either button updates the <h3 id="formTitle"> inside the popup and sets a currentType variable, indicating whether the transaction is an expense or income. The popup <div id="popupWrapper"> is then displayed.

When the form <form id="transaction"> is submitted, the default behaviour is prevented, and values from the input fields with IDs amount, category, description, and date are extracted. The category string is converted to uppercase to ensure consistency, and the code checks if the category already exists in categoryData. If it doesn't, a new category block is created using addCategoryBlock(category).

This function generates a new <div class="categoryBlock" id="block-CATEGORY"> and appends it to either the expenseCat or incomeCat section based on the category type.

The transaction entry is then pushed into the categoryData[category].records array, and the updateCategoryDisplay(category) function is called to recalculate the total for the category and update the display accordingly. Finally, the form is reset, and the popup is closed.

A screen shot of a computer program

Description automatically generated

**Implementation of the Category Colour Editor**

When a user clicks on a category block, the corresponding category is selected and the categoryPopupWrapper is opened. Inside the popup, I included three sliders: one for adjusting the hue, one for brightness, and one for saturation. Each of these sliders has an on input event that calls the applyColor() function. This function constructs a colour string (in HSL) based on the slider values and applies it to the background of the categoryPopup element.

When the user clicks the submit button (applyColorBtn), the color selected via the sliders is applied to the background of the selected category block (selectedCategoryDiv). The new colour is also saved into the categoryData object for that category, ensuring it persists. After applying the colour, the popup closes.

A screen shot of a computer code

Description automatically generated

**Key Functions:**

1. applyColor(): This function reads the current values of the three sliders (hue, brightness, and saturation), combines them into an HSL colour string, and applies this color to the background of the categoryPopup.
2. Submit Action (applyColorBtn): When the user clicks the submit button, the code reads the final values of the sliders, constructs the HSL colour string, applies it to both the selected category block and the category popup, and stores the colour in categoryData.
3. Category Block Creation: When a category is added, a categoryBlock element is created, which shows the category name and total spending. Clicking on a block opens the category colour editor popup and allows the user to edit its colour.
4. addCategoryBlock(): This function creates a clickable block for each category and appends it to either the "Expense" or "Income" section, depending on the category type.
5. updateCategoryDisplay(): This function updates the category block to show the current total amount spent in that category.

**Event Listeners:**

1. The sliders (hueSlider, brightnessSlider, saturationSlider) update the popup’s background colour live as the user adjusts them.
2. Clicking the submit button applies the selected color to the category block and stores it.
3. The category colour is saved in categoryData so that it persists even after the popup is closed.

This implementation allows users to customize the visual appearance of their categories, making it easier to track and distinguish between different types of expenses and incomes.

A screenshot of a computer

Description automatically generated

**Implementation of the Mood Module**

To start on the mood diary, I created a function called generateCalendar() that loops through the numbers 1 to 30. For each day, it creates a <div class="calendarCell">, assigns the day number to cell.dataset.day, and appends it to the #calendarGrid element.



A computer screen shot of text

Description automatically generated

Within this loop, I added a click event listener to each calendar cell. When a user clicks a cell, the following happens:

1. The clicked cell is saved as currentCell.
2. Any previously selected mood is cleared by removing the selected class from all mood buttons (moodBtns).
3. The mood selection popup (#moodPopupWrapper) is displayed.

The popup contains four mood buttons, each with a data-mood attribute and CSS classes for visuals like colour and emojis. I set up an event listener btn.addEventListener("click") on each button, so when a button is clicked:

1. The button is highlighted.
2. The selected mood (selectedMood) is set to the data-mood value of the clicked button.
3. All other mood buttons are deselected.

If currentCell (the clicked calendar day) and selectedMood are valid, it updates the content of currentCell to display the emoji and the day number, and sets the background colour to the corresponding mood colour. Finally, the popup is closed and the selection state is cleared.

The result is then: the calendar now displays mood emojis and their associated background colours for each day. The user can select their mood for a day, and it will visually appear on the calendar with both the emoji and the corresponding background colour.

*A screenshot of a computer

Description automatically generated*

*Resulting mood selection pop-up page*

**Overcoming Challenges**

During development, I encountered several bugs that helped me refine my debugging process. One of the issues was when the mood popup retained the previous selection, which led me to realize that I hadn't cleared the button states when reopening the popup. To fix this, I created a helper function resetMoodButtons(), which loops through the moodBtns and removes the selected class from each button.

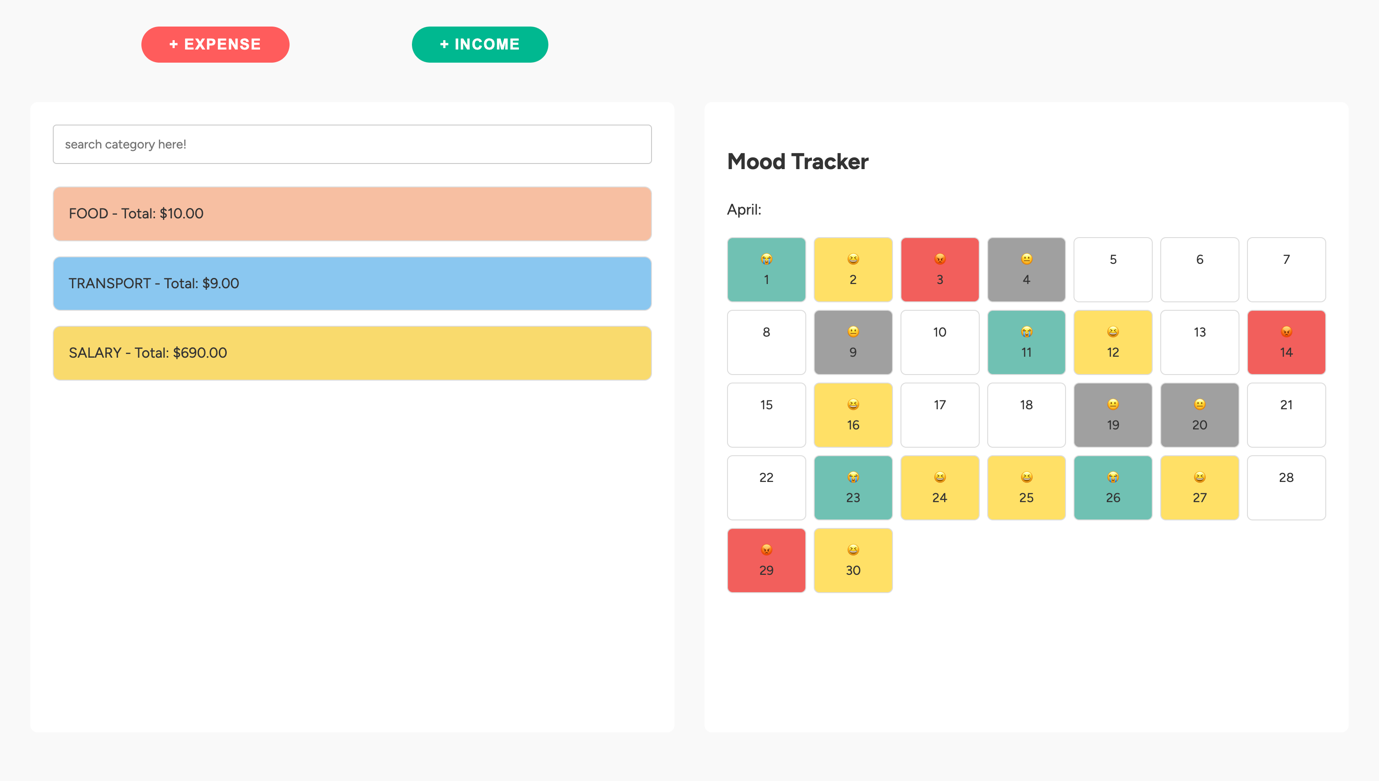
Another bug occurred when silent clicks on the calendar cells stopped working. I resolved this by moving the event listener for the calendar cells inside the loop where each cell is created, so that the listener was correctly bound to every calendar cell.

Additionally, I ran into errors when trying to apply stored moods before the calendar had been fully created. To fix this, I restructured the start-up sequence: first, I load the stored moods with loadMoods(), then generate the calendar with generateCalendar(), and finally apply the stored moods using applyStoredMoods().

**Final Outcome and Reflection**

The completed Spending + Mood Tracker is a single‑page application that lets me add income and expense entries, customise category colours, filter categories by search, and log daily moods with a single click.

Although mobile responsiveness and export features are future enhancements I would like to add, I take pride in how carefully each feature was built through small, iterative steps and how debugging with console logs and careful ordering prevented timing issues. Translating paper sketches into reliable code reinforced the importance of a clear load‑render‑bind workflow and honest experimentation with alternatives. This project deepened my understanding of interactive web development and prepared me to tackle even more complex interfaces in the future.

**

*Final Website Layout*