PROJECT DOCUMENTATION

INTRODUCTION

Description of my project:

- This project is an advanced expense tracker that tracks yearly expenses based on categories like food, entertainment, rent, etc.
- Under a designated budget, users can add new expenses and the app will indicate if they go over budget or have had a surplus amount left.
- **Visual Representation:** For each year, a pie chart will show where the budget (broken down into categories) is being spent and where users can reduce their spending.
- A line chart showing their budget, income, surplus, and expenses annually will be plotted. This will give users a better understanding of their spending habits.
- The technologies that I will use to build the app (e.g., **React, JavaScript, Node.js, Nivo, Firebase**)
- The features that I plan to include in the app (e.g., the ability to add and delete expenses, the ability to export data, etc.)
- **The target audience for the app** (e.g., people who want to track their spending, people who want to save money, etc.)

App Features -

- Expense entry: Users can enter expenses with their amount, year, description, and category.
- Expense chart: The annual expenses are charted on a pie chart, which helps users visualize their spending habits.
- Expense categories: Users can see their expenses broken down into categories, which helps them identify areas where they may be overspending.
- Income and budget: For each year, an income and budget is assigned. A line chart is used to track expenses, surplus, and budget.
- Expense deletion: Users can delete expenses from a year.

DESIGN

Dark Theme

The dark theme is a popular choice for websites because it can create a sense of calm and focus. The dark colors are less stimulating than bright colors, which can help you to stay

focused on the task at hand. Additionally, the dark theme can help to reduce eye strain, which is especially important if you are using the website for long periods of time.

Division of the Website

The website is divided into two parts: the left part and the right part. This division makes it easy to find the information you need. The left part of the website contains the pie chart and line chart, which show your expenses and income. The right part of the website contains the form to add a new transaction, the current selected year, and the transaction history of the selected year.

Use of Charts and Graphs

The use of charts and graphs makes it easy to visualize your expenses and income. The pie chart shows your expenses broken down into several categories, such as rent, bills, utilities, lifestyle, etc. The line chart shows the expense and income trends for all years. This information can help you to track your spending habits and make informed financial decisions.

Use of Responsive Design

The use of responsive design ensures that the website looks good on all devices. This is important because more and more people are using their smartphones and tablets to access the internet. Responsive design means that the website will automatically adjust its layout to fit the screen size of the device that is being used.

Use of Animated Elements

The use of animated elements adds interest and makes the website more engaging. The pie chart and line chart are both animated, which makes them more visually appealing. The animation also helps to draw attention to the data that is being displayed.

Font Family

I chose the Raleway font for my project because it is a modern and minimalist sans-serif font that is legible and versatile. The font's geometric structure and smooth curves make it a good choice for websites with a dark theme, as the font's thin lines will not be lost on

the dark background. Additionally, the font's versatility means that I can use it for a variety of purposes on my website, from headings to body text.

DEVELOPMENT

Todo List:

- [x] ~~Make the transaction form visible only when the "ADD NEW TRANSACTION" button is clicked and hide the form by clicking "CANCEL" or "ADD TRANSACTION"~~
- [x] ~~How do I store transaction data with year as key and the value is all the transactions that happened in that year?~~
- [x] ~~Using Firebase, fetch the data about transactions from the database, arranged in the year, and as explained above.~~
- [x] ~~After the above feature, use the year that is selected by the user and shows that data only, use Context API for this if necessary.~~
- [x] ~~Also, add the functionality to store the new data in the database~~
- [x] ~~Show transactions for the entire year but to keep it clean, use the scroll.~~
- [x] ~~When the delete icon is clicked, delete the transaction from the list and also reflect the same in the Firebase database. Complete the "DELETE" dispatch call for list context.~~
- [x] ~~When the year is selected, the pie chart of that year's expenses is shown.~~
- [x] ~~Also, on the selection of year, show that year's income, transaction, and balance~~
- [x] ~~Render the expense list after fetching data from the server.~~
- [x] ~~Use Context API, so that any new expense submitted (by form) is stored in the context object and I don't need to reload to get the fetched data. Also saves the number of fetch calls.~~
- [x] ~~Plot a graph showing year-wise expenses so that trends can be derived. Use a Line chart from Nivo for this purpose.~~
- [x] ~~Use pie charts to depict category-wise expenses made each year~~
- [x] ~~Surplus isn't showing, fix that, also style the line Chart~~
- [x] ~~The years in the drop-down must be displayed in sorted order.~~
- [x] ~~Show whether the transaction is an expense or an income by the border-right color change.~~
- [x] ~~Add the category tag to the Tile.js UI component also.~~
- [x] ~~Fix CSS of Tile.js as desc and amount are not consistently spaced.~~
- [x] ~~Add form validation and error classes to the new transaction form~~
- [x] ~~Add loading states for better UI/UX when fetching data.~~
- [x] ~~Use context API to transfer the year and expenses for that year to the expense list component and to the piechart.js~~

- [x] ~~Implement Error handling, for better UI, or if Firebase is down (even though that's unlikely - prepare for the worst!)~~
- [x] ~~Use prop-drilling to send selected year info to different parts of the component tree!~~
- [x] ~~Send the Expense value of a year to ExpenseState, calculated by iterating all the transactions for that year and calculating the net Expense, to different parts of the component tree!~~
- [ ] (Optional) Add the functionality to add a new year, with its own income and balance.
- [ ] (Optional) Consider refactoring the code in ExpenseForm.js
- [ ] (Optional) Allow deletion of a year State altogether, along with all transactions for that year.

Technologies used:

Firebase:
Firebase is a backend-as-a-service (BaaS) platform that provides a number of features for developing and managing web and mobile applications. These features include real-time data storage, authentication, and hosting. Firebase is a popular choice for developing React applications because it provides a simple and easy-to-use API for interacting with its features.

React:
React is a JavaScript library for building user interfaces. It is one of the most popular JavaScript libraries in use today and is used by a wide variety of companies, including Facebook, Instagram, and Netflix. React is known for its performance and its ability to create complex user interfaces.

Nivo:
Nivo is a charting library for React. It provides a number of charts, including bar charts, line charts, and pie charts. Nivo is a popular choice for React developers because it is easy to use and provides various features.

CSS Modules:
CSS Modules are a way of using CSS in React applications. It allows you to create custom CSS classes that are scoped to your components. This helps to prevent naming conflicts and makes your CSS code more maintainable.

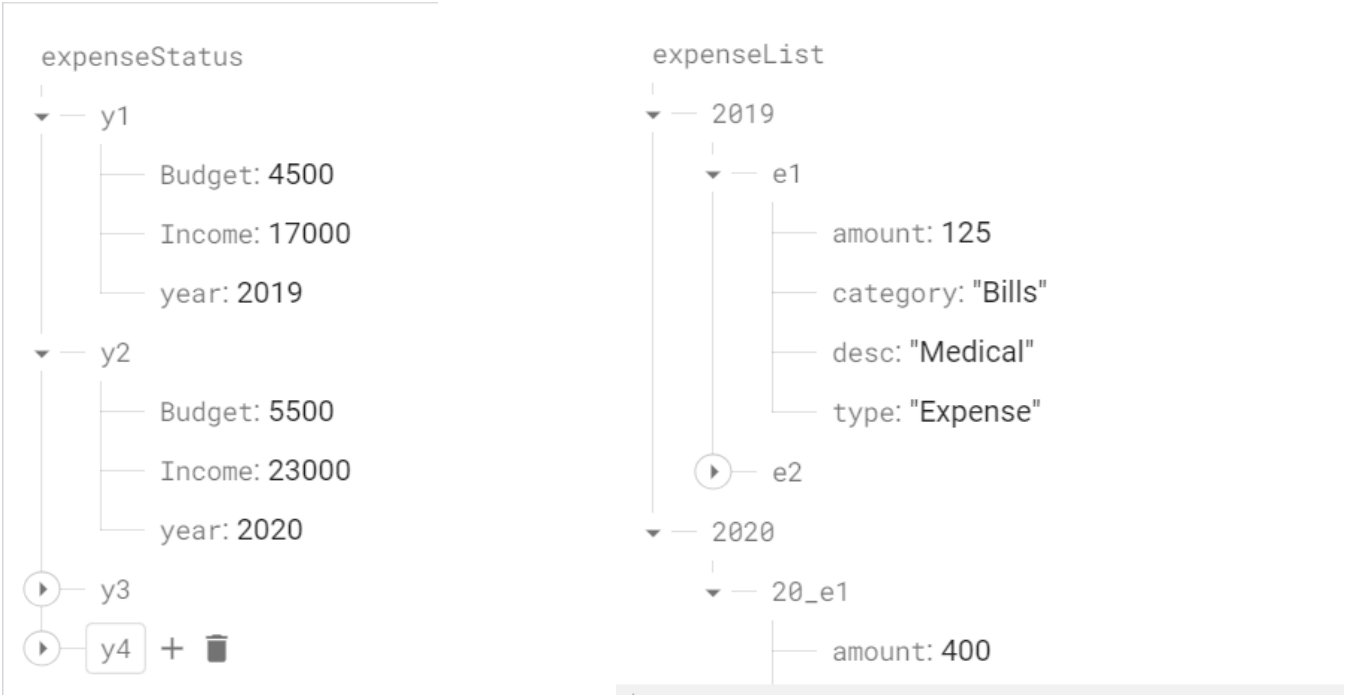
Node.js:
Node.js is a JavaScript runtime environment that runs on the server side. It is used to build a wide variety of applications, including web applications, real-time applications, and data-intensive applications. Node.js is a popular choice for React developers because it allows you to build server-side rendered React applications.

Create React App:

Create React App is a tool that helps you to create React applications. It provides a number of features, including code scaffolding, bundling, and deployment. Create React App is a popular choice for React developers because it makes it easy to get started with React development.

The Learning Curve -

1. **Learned how to fetch nested data from Firebase**, like the expenseList and expenseStatus, which is organized by year and each year has a number of transactions, each with a unique key, with a nested object that holds data about that transaction.

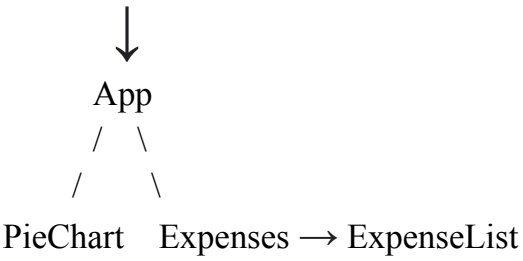



```
const responseData = await response.json();
      /* the data will be stored as follows ->
      An array of objects with two keys: year and transactions, the value
of year will be the year as a num
      and the value of transactions will be an array of objects, each
object holding each transaction.
      */
      const loadedExpenseData = [];
      for (const year in responseData) {
```

```
    const allTransactionsOfYear = [];
    for (const key in responseData[year]) {
      const transaction = {
        id: key,
        amount: +responseData[year][key].amount,
        category: responseData[year][key].category,
        desc: responseData[year][key].desc,
        type: responseData[year][key].type,
      };
      allTransactionsOfYear.push(transaction);
    }
    const yearData = {
      year: +year,
      transactions: allTransactionsOfYear,
    };
    loadedExpenseData.push(yearData);
  }
```

2. To get the selected year, we are going to **use prop drilling and lifting the state up** instead of context API, as we only are going to deal with this one small variable. It is also a nice way of showing different concepts of React and utilizing my knowledge of React. We will use a variable and pass it as follows :

App ⟶ ExpenseState ⟶ Lift the state up and assign value and initiate prop drilling



3. ***Create, Read, and Delete data from Firebase***: Using JavaScript's fetch API for data operations has been a powerful experience. GET and DELETE requests felt intuitive, but POST requests were initially challenging. With practice, I now appreciate its control and efficiency, enabling me to build dynamic applications effortlessly.

4. ***Data visualization:*** Using the Nivo charting library, I displayed my expense breakdown with delightful pie charts. They showcased annual expenses categorized into different spending areas, providing an engaging touch to my app. Additionally, I used line charts to depict expense trends over time, empowering users to make informed financial decisions. Nivo's versatility and interactive features made data visualization effortless and impactful, elevating the overall user experience.

5. ***Using React Features:*** During the process of building my expense tracker with React, I harnessed various powerful React features and hooks. I employed useState for state management, useEffect for handling side effects, and useContext for the centralized state. useRef came in handy for DOM interactions, while useReducer helped manage complex state changes. To optimize rendering, I utilized Fragments, and for seamless user interactions, I implemented two-way binding and controlled components. By lifting the state up and using the Context API, I prevented prop drilling and maintained a clean codebase. Lastly, I embraced CSS Modules for modular and maintainable styling. Together, these React concepts enabled me to create an efficient, scalable, and user-friendly expense tracker application.

6. ***Optional chaining:*** In my project, I made great use of optional chaining in JavaScript, and it proved to be a game-changer. This feature allowed me to handle deeply nested data structures with ease, avoiding those dreaded "Cannot read property '...' of undefined" errors. By chaining properties and methods with a question mark (?.), I ensured that if any intermediate value was null or undefined, the expression would gracefully short-circuit, returning undefined instead of throwing an error. This not only made my code more robust and concise but also improved the overall user experience by preventing unexpected crashes and glitches. Optional chaining has become an indispensable tool in my JavaScript toolkit, and I will definitely keep using it in future projects to streamline data access and enhance code reliability.

Major Challenges Faced and their Resolutions -

1. **Using context API to store allYearData inside ExpenseState :**

One of the most difficult steps yet was to send data to the context object correctly from a component that renders due to state changes and useEffect after first fetching data from Firebase.
You can't use yearCtx inside the above useEffect as it executes only 1 time on the first render. You can't use yearCtx inside loops or after conditional exit return statements (one is written below the second useEffect we use in ExpenseState).
You can't use it outside the first useEffect as yearCtx itself uses state management, so when ExpenseState component is re-rendering, we are trying to update the context object, which is not a valid operation.
Thankfully, google's Bard came to the rescue.

We will get the following warning -
Warning: Cannot update a component (`YearProvider`) while rendering a different component (`ExpenseState`).
So, the solution is simple, but requires in-depth knowledge of how React works!
We will first set the yearData and only after it is set, will we provide data to the yearCtx in another useEffect hook.

The rendering of the ExpenseState component and the update of the yearCtx context object work together in the following way:

1. The ExpenseState component renders for the first time.
2. The useEffect hook in the ExpenseState component runs and sets the yearData state variable
3. The useEffect hook also runs and updates the yearCtx context object with the latest yearData state variable.
4. The YearProvider component renders and displays the latest yearData state variable.
5. If the yearData state variable changes, the useEffect hook in the ExpenseState component runs again and updates the yearCtx context object with the new yearData state variable.
6. The YearProvider component renders again and displays the new yearData state variable.
7. This process repeats whenever the yearData state variable changes. This ensures that the YearProvider component is always updated with the latest yearData.

Here is a diagram that illustrates the process:
ExpenseState
useEffect
setYearData

yearCtx.loadYears
YearProvider
Render

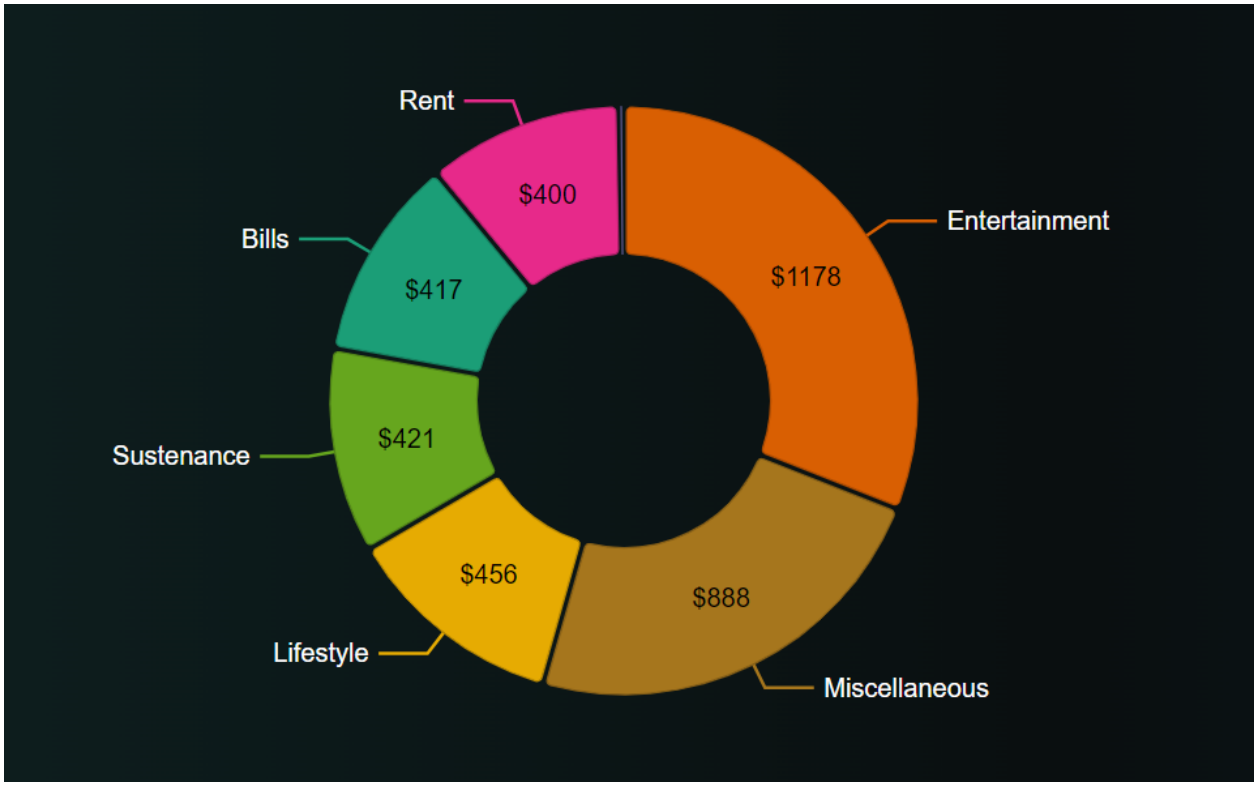
2. Rendering ExpenseList each time a new expense is added:

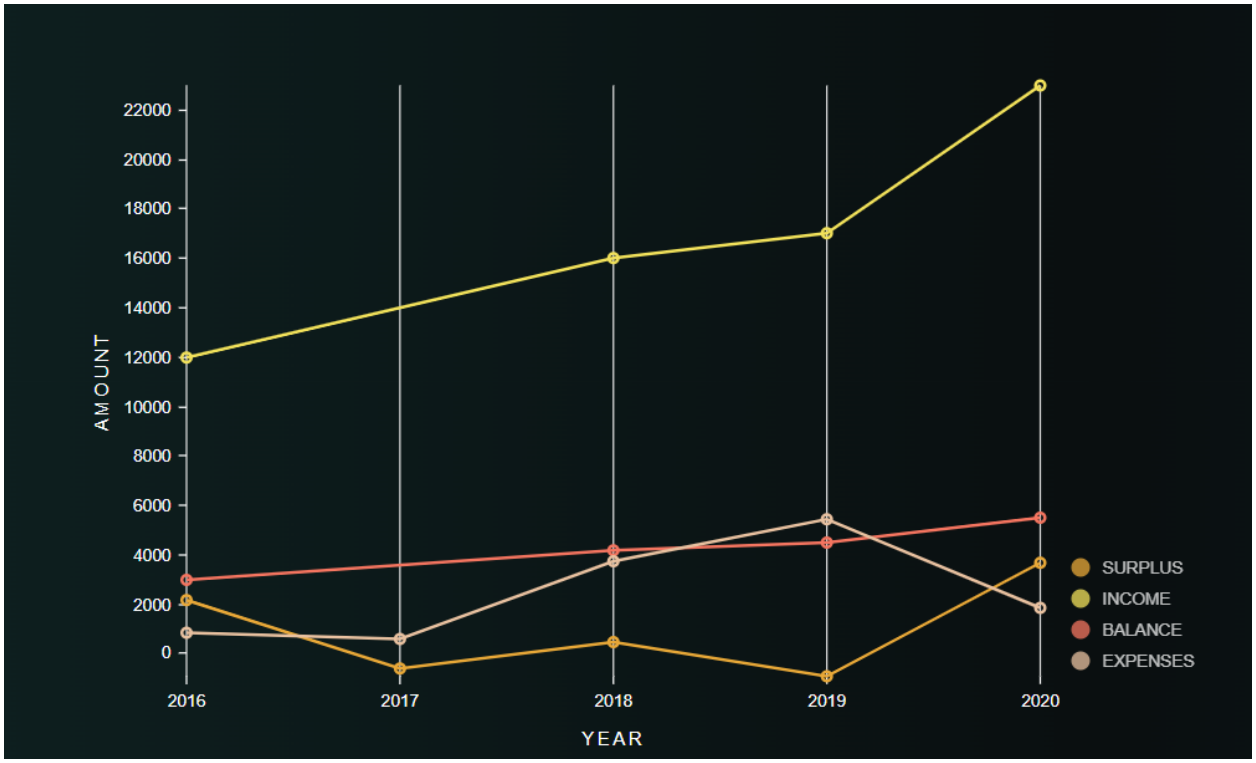
We are currently rendering expenseList by passing the fetched data down to ExpenseList component. The problem is that when a further expense is added, expenseList is not updated unless I reload the page, this can be fixed by using Context API so that any new expense is stored in the context object and I don't need to reload to get the fetched data. So, we will store the new data in the context object and use this object to render the list, on a reload of the webpage, new data will be fetched. This reduces the number of fetches we need to make from Firebase and makes our website efficient.

3. Using PieChart to depict Expense records for a selected year:

I want to use a pie chart in my React project, the pie chart should depict the expense amount that each transaction category is responsible for. There are a number of categories, namely, clothes, lifestyle, utilities, bills, miscellaneous, etc. For each year, I have a fixed income and budget set aside. I want to show this pie chart for each year, but one at a time, that only shows the pie chart of the year selected. The pie chart should be interactive and eye-catching.

After much deliberate research, I decided to use the Nivo chart library.


Rent
$400
Entertainment
$1178
Bills
$417
Sustenance
$421
$456
Lifestyle
$888
Miscellaneous


22000
20000
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
AMOUNT
2016
2017
2018
2019
2020
YEAR
SURPLUS
INCOME
BALANCE
EXPENSES

4. Using context API to store listExpenseData using useReducer hook :
I faced a challenge when trying to store the listExpenseData using useReducer hook. I wanted to be able to store the data in a way that was accessible to all components in my application, but I also wanted to be able to update the data in a way that was thread-safe.

I resolved this challenge by using the context API. The context API allows me to create a global state that is accessible to all components in my application. I then used the useReducer hook to update the state in a thread-safe way.

I used three dispatch action types, namely, ADD, LOAD, and DELETE. The Load action type fetches the data from Firebase and then we organize the data and set the allYearTransactions value to this data. It also calculates the expenseAmount for each year and stores it in allYearExpenses array of objects.

The ADD action type takes a new transaction as input, received after user fills the form to add new transaction and this data then is added to the specific year in the allYearTransactions. It also updates the allYearExpenses array to reflect new expense in the year.

The DELETE action type takes two arguments, transaction id (t_id) and year of transaction and deletes data about the transaction from both arrays.

The useReducer hook ensures thread safety by using a technique called latching. Latching is a way of ensuring that only one thread can access a particular piece of code at a time.

The useReducer hook ensures thread safety as follows-
When one uses the useReducer hook, it creates a latch object. A latch object is a special object that can only be accessed by one thread at a time. When one calls the dispatch function, the useReducer hook locks the latch object and then executes the reducer function. Once the reducer function has finished executing, the useReducer hook unlocks the latch object and allows other threads to access it.

This ensures that only one thread can be updating the state at a time, which prevents data corruption and race conditions.

5. Deleting a transaction from both the list Context Object and from the Firebase database.
I wanted to be able to delete a transaction from both the list Context Object and from the Firebase database. I wanted to do this so that the transaction would be removed from the list on the user's screen and also from the Firebase database.

I achieved this by ***listening for a click event on the delete icon in the Tile component***. When the icon was clicked, I would dispatch a DELETE action to my context object. This action would remove the transaction from the list.

I then made a DELETE HTTP request to the Firebase database. The DELETE HTTP request would remove the transaction from the Firebase database.

I made sure to make the DELETE HTTP request after the DELETE action was dispatched to my context object. This ensured that the transaction was removed from the list before it was removed from the Firebase database.

I couldn't use the DELETE request inside my dispatch call because the dispatch function is a synchronous function and the DELETE HTTP request is an asynchronous function. If I tried to use the DELETE request inside my dispatch call, the dispatch call would not return until the request had been sent to the server and the response had been received. This would cause my application to hang while the request was being processed.

To avoid this, I needed to make the DELETE request after the dispatch call had returned. This would ensure that the transaction was removed from the list before the request was sent to the server.

I did this by creating a function called **deleteTransaction**. This function would make the DELETE HTTP request to the Firebase database. I then called this function after the dispatch call had returned.

This approach worked well and allowed me to delete transactions from both the list Context Object and from the Firebase database without causing my application to hang.

6. Trade-off for simplicity:
I used the context object to reduce the number of fetch calls. When a new transaction was

added, I didn't need to fetch that from Firebase again and again. I only posted the data to Firebase and stored the data in my context object.

This saved me from making unnecessary fetch calls. It also made my application more responsive, as it didn't have to wait for the data to be fetched from Firebase every time the user wanted to view the list of transactions.

I'm glad that I was able to use the context object to reduce the number of fetch calls. It made my application more responsive and efficient.

But, this created an issue of its own!

One issue that I faced when using the context object was that the ids of the transactions were different in the context object and in Firebase. This was because I was using the useId hook to assign a random id to the transactions in the context object. However, Firebase assigns its own unique id to each transaction.

This meant that when I deleted a transaction from the list, in the same render cycle as when I added a new transaction, the new transaction would be deleted from the context object but not from Firebase. This was because the ids of the transactions were different. There are a few ways to solve this issue. One solution is to only post new data to Firebase just before the user reloads the page. This means that the data was never posted to Firebase, so there is no need to delete it from there. To achieve this, one can attach an event listener to the ***beforeunload*** event in the useEffect hook. This event is triggered just before the page is reloaded or closed. Inside the event handler, one can make the fetch call to the server.

Another approach is to use other properties (maybe all properties) instead of id to delete data from Firebase. This means that you would need to compare the properties of the transactions in the context object and in Firebase to find the matching transaction.

Here are some other alternatives to solve this issue:

- I could use a global id generator to ensure that the ids of the transactions are the same in the context object and in Firebase.
- I could use a database synchronization tool to keep the data in the context object and in Firebase in sync.

- I could use a custom hook to handle the deletion of transactions from both the context object and Firebase.

I decided to use the useId hook to assign a random id to the transactions in the context object. This meant that the ids of the transactions would be different in the context object and in Firebase. However, I was okay with this trade-off because I wanted to keep my application reactive and also because this is my first React project.

I'm confident that I can revisit this issue in the future and implement a more complex solution if necessary. However, for now, I'm happy with the decision I made.

DEPLOYMENT

I deployed my expense tracker using Vercel. Vercel is a hosting platform that is specifically designed for React projects. It makes it easy to deploy and manage React applications, such as automatic code splitting, hot reloading, and serverless functions.

Here are the steps involved in the deployment process:

- ➢ I created a Vercel account and linked my GitHub repo.
- ➢ I created a new project in Vercel and selected the Create from GitHub option.
- ➢ I selected my GitHub repo and clicked on the Deploy button.
- ➢ Vercel deployed my application to the cloud and I was given a URL to access my application.

I was happy with the deployment process. It was easy to follow the steps and Vercel deployed my application quickly. I was also able to access my application using the URL that was provided.

CONCLUSION

This documentation has outlined the design and implementation of a user-friendly and informative expense tracker website. The website uses a dark theme, Raleway font, and responsive design. The website also uses charts and graphs to visualize expenses and income. These features make the website easy to use and informative. The website can help users to track their expenses, visualize their spending habits, and make informed financial decisions.

I invite you to visit the website and try it out. The website is still under development, but I am confident that it will be a valuable tool for anyone who wants to manage their finances. I am also open to feedback and suggestions, so please feel free to contact me if you have any ideas for how I can improve the website.

——----------END OF FILE—----------