Froogal: Final Report for Stage 1 Development

By: Chris Adamson, Dylan Commean, Johnathan Dickson

Link to GitHub: https://github.com/jdcodes-12/froogal

Table of Contents

Why Froogal

The Problem

Financial literacy is the cornerstone of wealth building, it is a combination of many different financial concepts and tools that aims to increase one's IQ about money. An iconic, powerful tool that has been used throughout history is the budget. The budget's goal is to accurately track one's incoming and outgoing cash flow; however, an analysis only focused on incoming and outgoing values is not sufficient in and of itself. By only monitoring cash flow in certain categories, a key intangible metric is being missed - the "why." For example, say someone budgets and realizes he has spent $400 a month on coffee. To some, this may be okay but to others, this value could be alarming! The psychological motivations behind why he spent that much on coffee are unclear. Sure, he could just say "I'll spend less next month," but this mindset avoids digging deep into his spending habits. The problem with this mindset is that by neglecting to critically analyze his spending habits and only considering an arbitrary value, he never begins to form an understanding of his spending patterns.

The Solution

The lack of deep understanding of one's spending patterns is detrimental to the wealth-building journey. Without full awareness of who he is at the core, he will be unable to accurately dissect his spending patterns. This is where Froogal steps in as the all-in-one solution to reviving one's finances. Froogal aims to highlight one's spending habits by providing rich, data-driven visualizations that accurately track spending in a myriad of categories. Moreover, Froogal serves as a central repository for one's finances by tracking transactions, monitoring allotted budget amounts for varying periods (e.g. weekly, monthly, annually), displaying data models that react based on current spending and assisting in tracking due dates for expenses. By having all of these aspects consolidated in a central location, one can expect to: have better control over spending through accurate tracking of spending, gain a better understanding of spending habits by reviewing models rendered based on current spending data, and be provided with more detailed metrics that can help him understand his "why" in regards to spending. In other words, Froogal's main objective is to help one assess his spending by revealing who he is at the core.

Underlying Technologies

Frontend

React

What Is & Why React

According to the React official documentation, React is a JavaScript (JS) library for building user interfaces (UI). React embodies three unique philosophies towards crafting

beautiful, reactive UIs. The library relies on being declarative, component-based, and platform agnostic. The goal of React is greatly simplifying UI development while increasing code maintainability and development times. These advantages, described in more detail below, are the reasons the team chose to develop with React.

Unlike the imperative approach, vanilla JavaScript adheres to, which is explicitly stating when, where, and how some code artifact should behave or render, React utilizes a declarative method in building UIs. This approach allows the developer to state what should happen and/or how the UI should look at a certain point in time (based on state) and then React takes care of "how it should be implemented;" by using this approach, creating reactive UIs becomes substantially easier. React enables the developer to craft views for each state in an application by efficiently updating and rendering components whenever a component's state changes. This brings a more lively user experience (UX) and a dynamic, yet predictable UI which is easier to maintain and debug when needed.

What is a component? In React, components are the atomic building blocks of any UI whether they are reactive or not. Behind the scenes, components are JavaScript (JS) function calls that render some data visually; furthermore, components are not purely written in vanilla JS but instead are composed of a syntax, specific to React, called JSX. This syntax expands on JS but closely resembles HTML tags. By allowing developers to write code that looks similar to HTML but behaves like JS objects, a sharp increase in code readability and clarity is established. Moreover, how a component is written allows for the encapsulation of its state, logic, and visual rendering. Although slightly going against the separation of concerns, this approach to writing components makes it easier to maintain individual components within a UI. Additionally, since components are function calls, they can accept parameters (e.g. "props") which enable the developer to pass data throughout the application and separate the state from the document object model (DOM).

Lastly, similar to Java's mantra of "Write Once, Run Anywhere," React builds on a similar idea of learning the library once and then applying the code on many different platforms. For example, React code can be run on: mobile devices using React Native, can be run on the terminal using React Blessed, and on the server using Node.js. The library has been implemented to make no assumptions about the rest of the underlying tech stack of an application, but rather to focus on writing reusable code regardless of platform.

Learning Curve

React has been around for a long time and has undergone many drastic changes in how the library works (e.g. class components to functional components, rendering methods, etc.). Moreover, the ecosystem has substantially grown throughout the years. The pros of this are that many packages have been built on top of React which can be used to solve many use cases. The cons of this growth though, in the library itself and the ecosystem, is the learning curve. Getting started with React is rather straightforward if one has a solid foundational understanding of JS, however, mastering React is a different beast. For our team, learning React has not been the hardest challenge since most of the team has been previously exposed to the library; however, there were some concepts (e.g. state management and hooks) of React that, for some of the team, were challenging to understand and implement. Resorting to official documentation and online tutorials helped the team move past these hurdles.

Impact on Development

The utilization of React in Froogal cannot be understated. It is the cornerstone of Froogal since it addresses the need of crafting beautiful, reactive-based UIs. Without using React and using another method such as vanilla JS, development time would substantially be increased since an imperative approach to reactive UI design would have to be implemented. The imperative approach would have led to exponentially more lines of code being written. Additionally, it would have made it harder to manage the state, leading to more bugs in rendering the UI properly and accurately. Moreover, due to components' property of encapsulation, code maintenance is better supported by being able to easily monitor a component's logic, state, and visual rendering; thus, reducing the need for unnecessary files.

Chakra UI

What Is & Why Chakra UI

Cascading Style Sheets (CSS) is notorious for being time-consuming and a pain to write. Knowing this, and considering the time constraints in producing a prototype of Froogal, the team began seeking ways to greatly minimize efforts spent toying with vanilla CSS. After some research, Chakra UI was chosen as the styling framework for Froogal's front end. According to Chakra's official documentation, it is a component library that provides the building blocks for developing React applications. The library is straightforward, modular, and compact. To elaborate, Chakra provides a collection of predefined atomic components (e.g. Box, Heading, List, Flex, etc.) that can be glued together to compose larger, more complex components. It also offers larger pre-crafted components such as drawers, modals, dropdown menus, input fields, avatars, buttons, and more. Another advantage of Chakra UI is that it provides rich theming of components and design systems. By default, Chakra implements its base theme which is applied library-wide to components and design systems (e.g. color, typography, spacing, etc.); however, it also provides the ability to extend or override each of the default values by working with a configurable theme object (e.g. some JS object that affects the theming system). Additionally, another key advantage of the library, it focuses on accessibility (a11y). Each component comes with some degree of a11y baked into its default implementation.

Learning Curve

Chakra UI's learning curve is relatively low if one has already used some sort of CSS framework or library. Thankfully Froogal's team has had extensive exposure to CSS frameworks and component-based methods of crafting UIs. This prior experience made Chakra a breeze to learn and debug when needed. Moreover, the Chakra documentation is concise and well-written. The combination of prior experience and clear documentation enabled the team to overcome any stylistic hurdles that came up during development.

Impact on Development

Chakra UI played an essential role in developing Froogal's front-facing portions. By being component-based, it pairs well with the React way of crafting UIs. Additionally, by providing larger, commonly used UI components, such as dropdown menus, modals, input fields, etc., the time needed to craft these critical components were greatly reduced. By alleviating the need to create essential components from scratch, the team was able to spend more focus on

addressing more important decisions throughout development such as: what data should be collected from a user and how should that data be stored, how should the app manage state, and how should different components be rendered based on the app's state. Additionally, by providing components that adhere to good a11y practices, it has made Froogal more compatible with Web Content Accessibility Guidelines (WCAG).

Backend

Firebase

What Is & Why Firebase

Firebase is a robust platform that offers a variety of powerful, commonly used services in web and app development. It is built on top of the Google Cloud platform; essentially, a Firebase application is a Google Cloud project with extended, and optional, functionality and services offered through Firebase. Some examples of provided services are Firestore, Authentication, Storage, Hosting, Machine Learning, A/B testing, and Analytics. The platform offers various services but the ones that are essential to Froogal are Firestore, Authentication, and Storage. Due to time constraints, the team decided to explore the Firebase platform as a solution to Froogal's requirements.

Learning Curve

Although Firebase portrays itself as being an easy-to-use service, it is an enormous platform with many moving parts. Because of the sheer size and various services involved, the learning curve of Firebase is rather steep. Moreover, the documentation has been hit or miss for some of the services involved. Due to these factors, learning how to utilize and navigate the Firebase platform has been the greatest challenge for the team so far.

Firebase Authentication

What Is & Why Firebase Authentication

Firebase Authentication's objective is to reduce the complexity and time needed in crafting an authentication system. According to the service's landing page, creating an authentication system is resource intensive. One, it requires a large amount of focused time and effort. Two, after creating the system, a team of engineers is needed to properly maintain it. Built by the same developers that created Google Sign-in, Smart Lock, and Chrome Password Manager, Firebase Authentication allows small development teams to quickly setup a secure, robust authentication system, with a minimal amount of code, that leverages the experience of some of the brightest developers and handles complex cases such as merging accounts. Moreover, the service also provides customizable sign-in and registration solutions. The FirebaseUI, part of the service, contains various drop-in authentication solutions which handle different UI flows regarding user sign-in. These solutions are designed with best practices in mind and offer a variety of ways for a user to sign in or sign-up (e.g. Google, GitHub, Facebook, Twitter, etc.); thus, providing a more pleasing and modern onboarding experience. Due to the

team's lack of experience in building effective authentication systems, we decided it was best to leverage this service.

Impact on Development

Firebase Authentication plays an important role in Froogal's backend. Before a user can begin interacting with the dashboard, he must register or sign in to his account. This security check benefits Froogal by restricting access to the dashboard for unauthorized users and properly pulling in the correct data for a user that is signed in. Without leveraging this service, a large amount of time and effort would have needed to be devoted to making an authentication system, which likely would be subpar in terms of best practices and recommended guidelines. Furthermore, a user must be properly verified so the dashboard correctly renders the user's data; rendering improper data would defeat the entire purpose of Froogal.

Firebase Firestore

What Is & Why Firestore

Firebase Firestore is a service that provides the means of crafting serverless, secure apps which can easily scale. It does so by storing app data in the cloud, which then can be synced to offline and online devices, and can be queried expressively. Like any database, Firestore cannot be left unprotected. To solve this issue, Firestore enables the developers to write Security Rules that can give access to or restrict certain users from making requests to the database. Firestore follows the NoSQL model of storing data. It has two main components that make up the database's internals: collections and documents. To elaborate, collections are a list of documents. A collection can contain zero to many generated documents, which can be given specific or randomly generated identifiers. Documents are atomic units inside a collection that house all data related to that document. For example, in the database, there could be a "user" collection that would hold the document of "user#1239." This document would contain all the information related to user#1239. A document's information is stored in the form of JavaScript Object Notation (JSON), which is heavily used in many applications to efficiently pass data.

The design of Firestore's internal structure allows developers to create various, expressive queries on stored data. Providing methods that query and retrieve data from collections and individual documents in the Firestore opens opportunities for developers to provide a better user experience by requesting the data that is needed and also requesting additional data if needed to add to the UI's rendering, thus making a user's experience more customized to him.

Impact on Development

Firestore is the brain of Froogal's backend. Without this database, there would be no way to persistently store user data. This would mean that the dashboard would always render null state values and not accurately present a user's information. The user would not be able to store or keep track of his expenses, sums of transactions, allotted budgets, and total spending over various periods. By providing a means to easily store data in a NoSQL-like manner, it has allowed the team to focus less on relationships within the database by encapsulating all user data in a single document. This has made querying for certain pieces of user data easier due to not having to jump around many tables. Additionally, since the querying capabilities of Firestore are

robust, it has allowed the team to only query data that is needed instead of being sent back a gigantic JSON object. This reduces load and renders time, thus providing a better user experience.

Firebase Cloud Storage

What Is & Why Cloud Storage

Firebase Cloud Storage is a service designed to help easily integrate user-generated content (e.g. photos & videos) into an application. Cloud Storage exposes methods that enable developers to effortlessly store this kind of content. Furthermore, the Firebase SDK for Cloud Storage keeps mobile connectivity in mind. When app users are offline, reads and writes to the Cloud Storage are paused and then resumed once connected to the internet again. This functionality saves two precious resources for the user - time and bandwidth. Additionally, the SDK also provides ways of protecting this user-generated content that is stored. Just like Firestore, Cloud Storage is guarded by robust security rules developers write to restrict or permit access to the storage buckets containing the data.

Impact on Development

Cloud Storage serves only a single purpose in Froogal. The service is responsible for allowing a user to store an avatar, which is some picture or graphic, that represents the user. The original idea for this feature was to make the dashboard more personalized to the user, in hopes to increase user satisfaction; thus, encouraging frequent use of Froogal. After implementing Cloud Storage, the benefits do not seem worthwhile. For example, Froogal only holds user avatars in the Cloud Storage, this is a stark contrast to companies like Instagram or Facebook which need to store loads of user-generated content. Froogal may remove the ability to add profile avatars in the future.

External Packages & Other Tooling

Bcryptjs

What Is & Why Bcryptjs

Hackers and other malicious actors have discovered more advanced and efficient techniques to acquire user data. Because of this threat, there is a need to strongly secure sensitive user data such as passwords. Unfortunately, single unique passwords per user are no longer a best practice. This is because if a hack discovers someone's password and another user has the same password, that hacker then has access to all users with the discovered password; thus, resulting in increased security breaches, stolen identities, etc. To help combat this threat, the team decided that we use "hashing and salting."

The "hashing and salting" aims to protect user passwords by hashing passwords once, then adding some "salt" to an arbitrary place inside the hashed password. Salt could look like "lkjhasdi" appended to a user's password; the key is that the salt is only known to the server, so hackers are not able to discover this salt easily. This approach makes user passwords more unique, thus increasing their strength. Hashing and salting can be cumbersome to implement. To

reduce the complexity of implementing this approach, Froogal leverages the Bcryptjs package to add another layer of security for user data.

Recharts

What Is & Why Recharts

A financial app that didn't show any charts or signals and only showed numbers would be quite underwhelming. Fortunately, Froogal has kept user experience as a primary consideration during the design and development process. To create a more immersive dashboard, the team decided that data visualization was essential. However, data visualization is a notoriously hard task to implement due to the number of animations needed. To solve this challenging task, Froogal heavily depends on the Recharts package.

Recharts is a component-based data visualization library that exposes predefined, customizable components. These components vary in complexity, but some of the components offered are bar charts, radar charts, line graphs, pie charts, and shapes such as polygons, curves, etc. Lacking animation experience, but understanding good animations was critical to Froogal, the team decided to leverage Recharts API to provide visually appealing, reactive, data-driven components to the users.

Froogal's System Design

High-Level Overview

Froogal's system can be dissected into three main elements: the dashboard, sign-up and registration forms, and the Firebase backend. Both the dashboard and registration forms contain major components that compose the UI for that route. To see visuals of both routes, reference D1 and D2 in the appendix. This section elaborates on various components for the dashboard and sign-up routes, since Firebase was previously discussed it will not be mentioned again.

Dashboard

Froogal's dashboard is composed of nine major reactive components. Each of these components can be placed into one of the following categories: watchers, charts, lists, hubs, and settings and navigation. The subsections below introduce each component within the respective category and highlight the significance of these components within the dashboard.

Watchers

Watchers are a subset of the major dashboard components that are responsible for acting as visual indicators to the user. This category is composed of three different components: the OverUnderWatcher, the BudgetWatcher, and the TotalSpendingWatcher. In comparison to other components within Froogal, these are the smaller components. Despite their size, these components are still essential because they produce some value, which is based on user spending

habits, that the user can quickly glance at to assess their spending over a certain time period (i.e. weekly, monthly, annually).

The OverUnderWatcher's responsibility is to notify users about their current spending status when considering their respective allotted budgets and total spending for a period. For instance, if a user allocates $1,000 dollars for a monthly budget but spends over that amount, then the component will render a red arrow and the text "Over;" this allows the user to quickly glance at his current spending status within a period. Moreover, the component is designed to generate a random financial, inspirational quote each time the user logs in. The purpose of the quote is to encourage the user to keep practicing frugal habits.

The BudgetWatcher, as the name suggests, is designed to set and monitor a user's budget for some time period. A user's budget initially starts at the null state of $0.00; however, the user is able to adjust this value in two ways. One, the user can click the component's button which renders a modal that allows him to input a new budget. Changes here are saved properly into the Firestore and then reflected immediately on the dashboard due to React's rerendering functionality. Two, the user can utilize the FinancialDrawer to adjust his budget as well, more on this component in the Settings & Navigation subsection.

The TotalSpendingWatcher's purpose is to track all user spending over a certain duration. As a user generates more receipts in his profile, the component's value changes to reflect the sum of all receipt totals within that duration. Like the BudgetWatcher, the TotalSpendingWatacher also has a null state of $0.00. The only way to change this value is through receipt creation. Additionally, the component depends on the user's allotted budget for that period. It uses this data to render text that notifies how much the user has left to spend. For example, a user allots $250.00 for weekly spending. He then spends $100.00 for groceries. The TotalSpendingWatacher will deduct $100.00 from the $250.00 and then render the difference in the TotalSpendingValue's notification text.

Lists

Lists are another subset of major component categories. They are responsible for holding a collection of user-generated objects. In Froogal, there are two main types of user-generated objects: receipts and expenses. On the dashboard, the user will find the RecentReceiptsList and the ExpenseList; each is responsible for containing and displaying their respective user-generated objects in a visually pleasing manner.

The RecentReceiptsList's is designed to be the central place that a user can view their receipts. The list holds ReceiptItemCards which display overview details about individual receipts. On the card, a user can expect to find tags associated with the receipt, the date and location of transaction, total items and total price. Moreover, when a card is clicked a modal is displayed to the user. This user provides even more insight into a receipt's information by displaying all of the previous information as well as the list of items that compose the receipt. An important thing to note about the RecentReceiptsList is that it will only render the three most recent receipts. The newest created receipt will be rendered at the top of the list, while older receipts are pushed to the bottom. This is to prevent UI overflow which would distort the dashboard's layout.

The ExpenseList serves as the focal point of examining a user's expenses. Like the RecentReceiptsList, it also is composed of cards. These ExpenseItemCards briefly display data relating to an expense such as: the due date, the price, and the status. The status is one of the following enumerations: PENDING, PAID, or OVERDUE. If an expense is overdue, a text reminder will render on the card, notifying the user that the expense is overdue by some variable amount of days. Moreover, these cards function similar to those of the RecentReceiptList. When clicked, the expense cards also render a modal which displays more details about the underlying expense.

Charts

Charts are one of the most critical major component categories in the dashboard. They are responsible for visualizing arbitrary spending data so that the user can concretely see their spending. The visualizations allow the user to get a more clear picture of how their spending really looks; thus, providing deeper insights about personal spending habits. The dashboard contains two main charts: the CategoryBreakdownChart and the BudgetComparerChart. Each chart tracks and renders different data but aims to provide an in-depth look at spending habits through robust visualizations.

The CategoryBreakdownChart's purpose is to correctly distribute and render spending amounts of certain categories associated with the user. Its data visualization is rendered based on a user's receipt and tags collections. It uses receipts to reactively distribute the correct values to the chart, which causes the chart to be drawn in a way that highlights the user's spending. It uses tags to populate categories on the chart. The chart comes with some predefined, common categories such as: rent, food, clothing, etc. Additionally, a user can render more categories on the chart by creating more tag categories in their account. Spending will then be distributed correctly to those categories by pulling from receipt data.

The BudgetComparerChart's responsibility is to connect a user's budgeted amount and total spending for a certain period. The chart, which pulls from a user's receipt data, renders the correlation between the user's budget and total spending at the current moment. For example, a user sets a $100 dollar budget, then spends $50. The line on the chart which represents total spending will then reactively move to the correct data point. Once the total spending goes over the allotted budget for the period, the user can visually see they are spending more and are notified on the OverUnderWatcher. Moreover, the BudgetComparerChart adjusts for 7-day periods. It updates the days dynamically, thus allowing the user to correctly see their spending over a weekly period. Currently, the annual and monthly visualizations are still in testing.

Hubs

Hubs can be defined as a central area of focus or activity. On the dashboard, there is only one hub - the ReceiptHub. This hub serves as the main place of inputting user data when on the dashboard. Its operations consist of creating receipts, deleting receipts, modifying receipts, and searching for receipts. Then ReceiptHub is essentially a factory for Receipt objects. Everytime

the receipt hub modifies a user's receipt collection (e.g. creating a receipt, deleting, etc.) those changes are reflected in the dashboard. For instance, creating a new receipt will add the receipt's total price to the TotalSpendingWatcher, be added to the RecentReceiptsList, affect the BudgetComparer chart and alter the CategoryBreakDownChart. Conversely, deleting a receipt has the opposite effect. Without the ReceiptHub, the user would not have means for data input, thus consequently making the application useless.

Settings & Navigation

Dashboards come in many different styles in terms of looks and functionality. Since Froogal is a Single-Page Application (SPA), thanks to React, it's important to contain everything in one place. Initially, Froogal's idea was to be implemented using a handful of HTML pages and then serve those individually; however, after realizing that serving a small amount of HTML to capture user and financial settings was inefficient, the team decided to bake routing into the dashboard itself. The components in the settings and navigation category, have been designed to routing as minimal as possible. In this category there is: the Sidebar, FinancialInfoDrawer, and UserProfileInfoDrawer.

The Sidebar is the only source of navigation within the dashboard. It contains: a button to open the FinancialInfoDrawer, a button to open the UserProfileInfoDrawer, a button to sign-out of the dashboard, and toggler which swaps between dark and light mode. Other than housing these buttons, the drawer does not have any further functionality. It is worth mentioning that the Sidebar is mobile responsive; however, some visual cleanup is needed to prettify the UI on mobile devices.

The FinancialInfoDrawer, as the name implies, is where the user updates their financial information. In this drawer, the user can set and update their allotted budgets and income for weekly, monthly, or annual periods. Additionally, the user can access all of their receipts and expenses, and swap the "finance mode" of the dashboard. Swapping the finance mode renders data on the dashboard in terms of the time period chosen (e.g. weekly, monthly, annually). Changes made in this drawer are reflected on the dashboard and the new data is sent to the Firestore when saved.

The UserProfileInfoDrawer is the control center for modifying user settings. In this drawer, a user can edit: their first name, last name, email, and password. Also, the user can add an avatar if desired. Changes that take place here are stored to the Firestore, overwriting the old data for the respective user document. The changes that take place in this drawer affect mostly the sign-in and registration route, since passwords have been modified.

Login & Registration

Froogal's login and registration route is composed of two primary components: the SignUpForm and the LoginForm. Each of these forms follow modern UI patterns and implement

common validation best practices. To see some of the validation messages, refer to D4 in the appendix.

Sign Up Form

The SignUpForm is one of the two forms used in the initial route of the app. Like most modern onboarding experiences, this form is used to create and add new users to a database. In Froogal's case, the user is authorized using Firebase Authentication. Once authorization is successful, the user is routed to the dashboard. There, the user is presented with a null state dashboard. Moreover, the SignUpForm is designed with additional layers of security as well. It uses regex to enforce the user to craft better passwords, has regex to make sure a name is the proper length, and validation to make sure emails are of the proper format. Moreover, the form has some baked in accessibility by leveraging Chakra UI. It's keyboard navigable by using tab, focus rings appear around focused inputs, and there is a "show" button which aids the user in creating passwords, since the initial password content is replaced with bullets.

Login Form

The LoginForm is very similar to the SignUpForm, it is just slightly scaled down. It does not require as many inputs to route the user to the dashboard. However, this component is not effective unless a user already has an account authorized. This is because using this form requires a valid authorization account to be already created in Firebase Authentication. If a user attempts to login without an existing account, he will be unable to proceed to the dashboard and will be notified that the account doesn't exist or is already taken. Moreover, the form has similar email validation to the SignUpForm's validation. It's important to note that the LoginForm does not care if a user has information in the Firestore. A user can be authorized and have never touched created any objects on the dashboard, this is okay since the form only relies on existing authorized accounts.

Minor Components

The minor components of the dashboard are atomic components which when put together compose larger major components. Currently, there are three categories of minor components that are embedded into the dashboard - Togglers, Modals, and Buttons. The following sections briefly describe each category, their components, and why those components were implemented.

Togglers

Togglers are components which switch between two values - on/off, true/false, show/hide - of some setting in the dashboard. As of now, there is only one toggler used in the application, this component is the ColorModeToggler. As the name implies, it's responsibility to render either dark mode or light mode. If the current mode is light, it swaps to dark and vice versa. This is not a necessary component for functionality, but it was implemented to help improve the user experience (UX) of the application as a whole.

Modals

Modals are one of the primary minor components. Unlike togglers, they are more prevalent in the application. The dashboard contains two kinds of modals - TileModalContainers and ButtonModalContainers. TileModalContainers are used for elements on the dashboard that need modal popups without buttons being rendered. ButtonModalContainers are used for elements on the dashboard requiring modals to be rendered with Buttons. One important thing to mention about these modals, is that they are wrappers around a ModalContainer component. The ModalContainer is responsible for layout a modal in the application. It takes a title, a size, body content (in form of a component), and handler functions to handle closing and opening the modal. These parameters are passed down from the wrapper components (e.g. Button and TileModalContainer).

Getting Started With Froogal

Firebase Hosting

The first step to hosting a Firebase project is to have a Firebase Account. After setting up your project, the user will want to host their application. The user will navigate to their command prompt and the root directory of their firebase project will run “npm install -g firebase-tools”. This will allow the user to access the Firebase Command Line Interface (CLI) to get started on the hosting process. The next step in the process, while inside your root directory, is to sign into the user’s Google account associated with the project using the command “firebase login”. After logging in, the user will enter the command “firebase init” to initialize their Firebase project. After initializing the user’s project, the user will need to register a name for their application in the firebase Console. The user will be prompted to install and add the firebase SDK to their application. They will be given configuration files to allow their app to hook up to Firebase. The user will need to run the command “npm install firebase” in the root directory of their project. The last step in the process is to run the command ‘firebase deploy’ and your application should be hosted on firebase. A link to the hosted website will be available in the command line as well as their Firebase console online.

Features, Not Bugs

Known Bugs

Steps have been taken to reduce as many bugs as possible in the application. Bugs are inevitable in a system regardless of the steps one takes. As programmers, we cannot reduce the number of bugs in our system to zero. Language constraints or other issues with the meshing of frameworks and engines are bound to cause issues. First is the visual bugs where containers are not lining correctly or other odd visual quirks. There is a visual bug on the sign-in and registration page for screen sizes that are too small. The page when viewed will stretch across the screen and obscure the typing in any of the inputs. The page is responsive in a way. At some

point, the containers for the sign-up and sign-in will appear on top of one another. In doing so, the sign-in page is no longer visible at the top and cannot be scrolled to. These are the only known bugs for the sign-in and registration page.

For the dashboard, there are some issues with the sidebar buttons. If a user doesn't click on the text that is within the buttons, the button will not work. Also on the dashboard, the layout is responsive and will transition to a mobile-friendly view when scaling the web browser window down to the size of a smartphone screen; however, when the user scales up to desktop size. The whole window disappears and will not come back unless the user refreshes the screen.

Receipts

Currently, there is no vetting of information in the receipt creation process. When creating a new receipt, there is a chance bad information can get in and crash the application or have unintended state consequences. This could happen if there is an incorrectly formatted date for the date field inside our receipt creation modal. When a user creates an incorrect date object, it could display a date that is 52 years ago which is only a visual bug. Firebase and JavaScript have different ways of handling Date objects. There is a function when our application is dealing with a Firebase date that the application calls and changes the date given to a JavaScript date object. When handling data on the front end, if our application accidentally calls this helper function on a Date object, our application crashes.

Unintended state side effects could also occur if a user creates a receipt without creating an item associated with the receipt. When creating a receipt with no items, the receipt data visualization will display "Not a Number" (NaN) in the field for the given date. It will also display NaN for the spending total. Our application relies on correct information being put into the receipt creation modal in some places. There are safeguards in place like the receipt item creation modal which vets the information coming into it. The button for item creation cannot be pushed unless certain conditions are met.

With the search of receipts, there is a problem where the application is supposed to show the receipt a user clicks on in the search modal for the receipt hub. The receipts populate inside of the receipt search modal; however, the application does not allow the user to click on a receipt and instead doesn't do anything.

Rendering

When rendering user information there will always be something that our team is unable to handle. These rendering issues happen primarily when it comes to rendering the receipt information. If a user types in a long enough string for any of the information such as the name or location of the receipt, the rendering of that information will push and maneuver the containers around into incorrect positions.

In the data visualizations, there are some issues when it comes to receipts that have an incredibly large amount. The data values cannot be visualized and will show a line on the cart that trails off the chart. Some error messages appear in the console when the category breakdown chart does not have values in it. I don't believe this will crash the application; however, it could in the future. To see an example of this visual bug, see Figure D4 in the appendix. Another data visualization visual bug occurs on the category breakdown chart. When a category name is too big for the chart, it does not automatically fit the name into the container of the chart. This

happens primarily on the default categories such as “Transportation” and any other user-generated category whose name is long. This bug is easy to miss when the category breakdown takes up the whole screen in our responsive design.

Expenses

Expenses currently do not render in the expense watcher component. The expenses inside of the component are hard coded to test for the rendering functionality. When clicking on an expense, it does not take you to the specific expense a user clicks. It instead takes them to a hard-coded example. The expenses as well have visual quirks depending on the length of the names. The containers will push information out of the way and out of order.

Froogal’s Future

User Interface & User Experience

Apart from what makes Froogal separate from our competitors is our outlook on the future and our dedication to always improve. One of the things that Froogal plans to focus on in the future is to drastically improve our user interface and our overall user experience when using our application. One feature that we would like to implement in the future is the dark mode interface for our product. Currently, the feature does work in theory, however, there are many instances where the feature is not as pleasant to the eye of the user. An example of this would be some of the charts and visualization of the graphs. We would love to improve on this feature and make sure that it is fully functional and that our product is easier on the eyes of our users.

Another major feature that we would like to implement in the future that would greatly improve the user experience of our application is to make sure that our application is responsive. At the moment, the application is semi-responsive, similar to the dark mode feature, however, many bugs are preventing the application from working as intended. Examples of this include the signup/login page components visually going off of the display whenever the screen is not a desktop computer. There are also minor issues in terms of the responsiveness such as various components on the dashboard not as visually pleasing on different sized screens. Making the application fully responsive will greatly improve the user experience in the future as users will have the capability to view Froogal from any device. This can range from smaller screens such as phones to much larger screens such as ultra-wide monitors.

Fixing some of the other bugs that are within the application is also a main focal point in terms of the future of Froogal. An example of one of these bugs is fixing and improving the sidebar. To be more specific, we would like to make sure that when a user clicks on a button within the dashboard, it will be able to execute. As of right now, you must click on the text of the button. Similar to what is already stated about the responsiveness, we would also like to improve this aspect on the sidebars and drawers as well. This improvement will help further influence and improve the user experience of Froogal.

Other improvements in the future that should be expected to see are additions to various animations throughout the components. This can vary from straightforward animations to complex ones, however, the main goal of these animations is to improve the overall experience for the user. If the implementation of the animations allows the users to find different features or

allow them to view our application better, then that is something that Froogal wants to aim for and achieve in the future.

Collecting Feedback

One of the major ways that we would like to collect feedback is by letting users be able to use Froogal and request new features that they think should be added to improve Froogal. Through this process, the users would also be able to point out different bugs and find potential problems that have not been discovered while using our application. This can lead to us being able to improve and fix potential bugs that were previously undiscovered and unknown. Through the use of this beta testing, Froogal would be allowed to create and produce a product that is more refined and complete for our useres.

Another way that we can collect feedback on Froogal is to create and administer tests internally. There would be tests that would test each of the components that we have on Froogal, and through those tests, we would collect data and feedback from the system to make sure that the components are working as intended. We could also collect data and feedback from the Firestore services and attempt to find innovative ways to decrease the numbers on data retrieval.

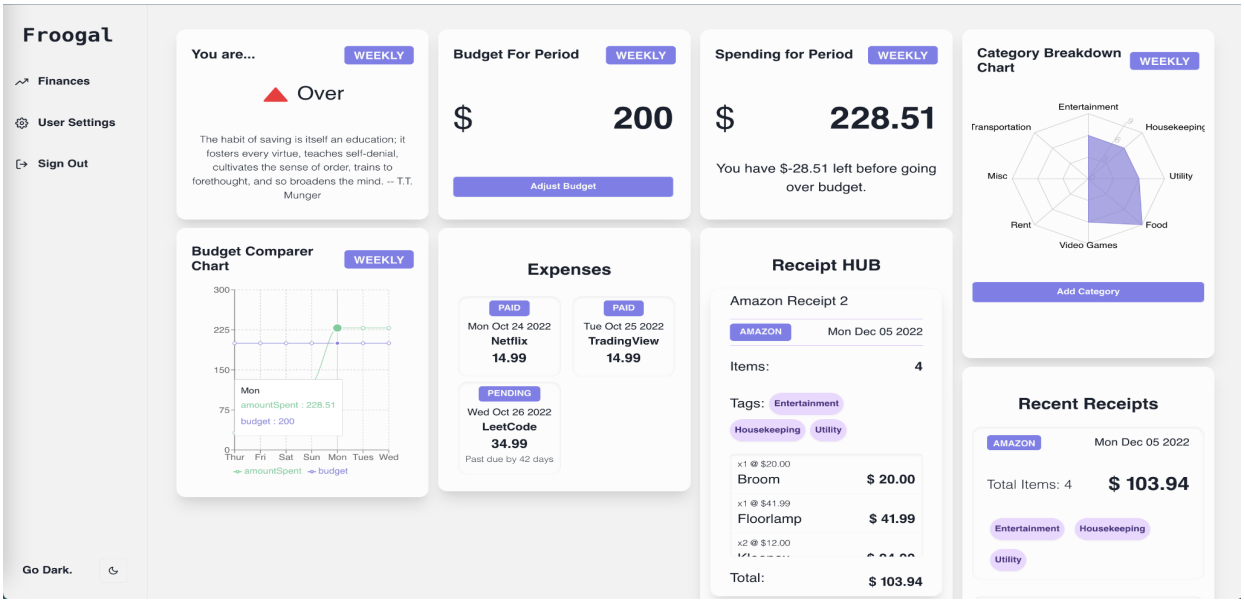
Implementing New Features

At Froogal, we are always trying to strive towards innovation and bringing forth new features that will allow our application to be the best in the market. With that being said, we have many features that we look forward to developing and implementing onto Froogal in the future. One of the many features that we have planned is that of emailing our users whenever one of their expenses are near its due date. This email notification feature will allow our users to be aware and assist them in avoiding late or missed payments to their expenses. This feature follows our core values at Froogal where we are trying to help our users to be Froogal, and to avoid misuse of their money such as paying unnecessary late fees to their expenses.

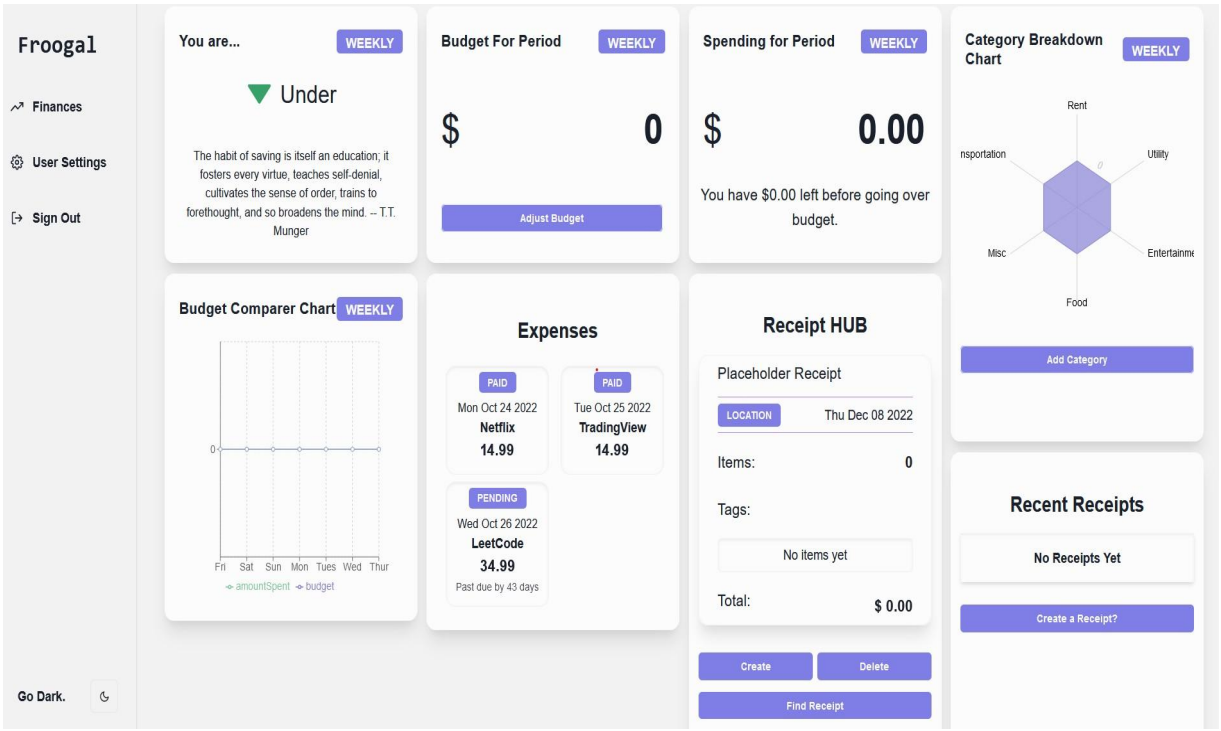
Another set of features that we would like to implement is to let the user have a more customizable control over the dashboard. An example of this would be to allow the users to see their data in a multitude of different graphs. For example, a user could see their category breakdown from a choice of graphs such as a bar chart, pie chart, radar chart, etc. Users could also add and remove different components from the dashboard as they please. For example, if a user feels as though the total spending tracker is not useful to them, they can simply remove it. The user always will have the option to read any component that they remove as well. This feature will essentially allow the user to have more control of their dashboard and allow them to have more control on their spending habits.

Appendix

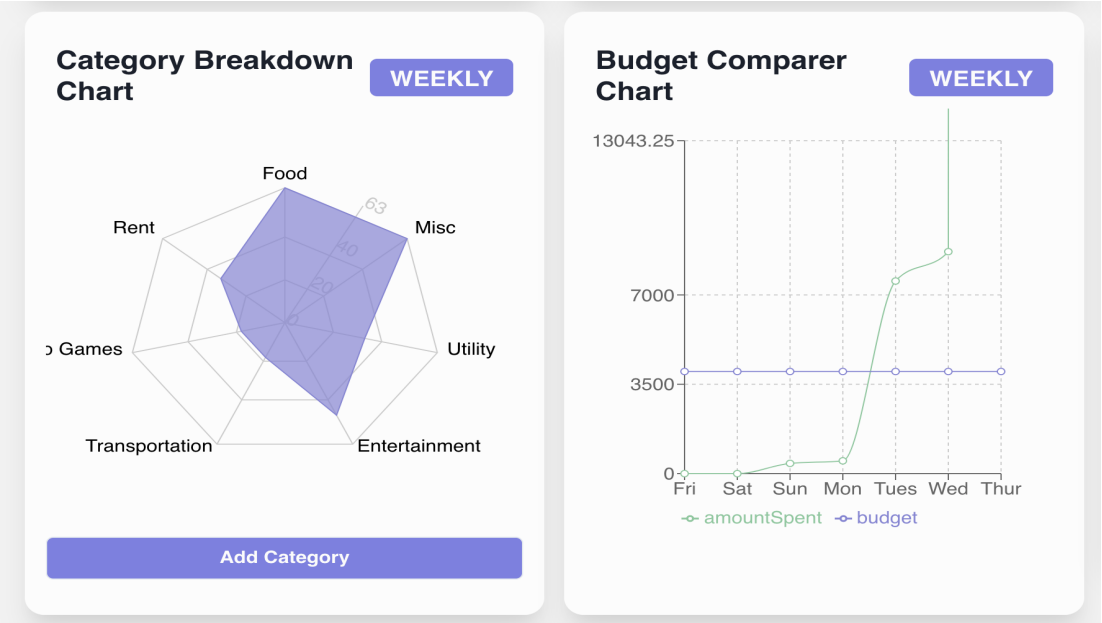
D1: Screenshot of Dashboard Route for an Existing User (non-null state):



D2: Screenshot of Dashboard Route for a New User (null state):



D3: Screenshot of the Visual Rendering Bug:



D4: Screenshot of Login Page Registration Errors