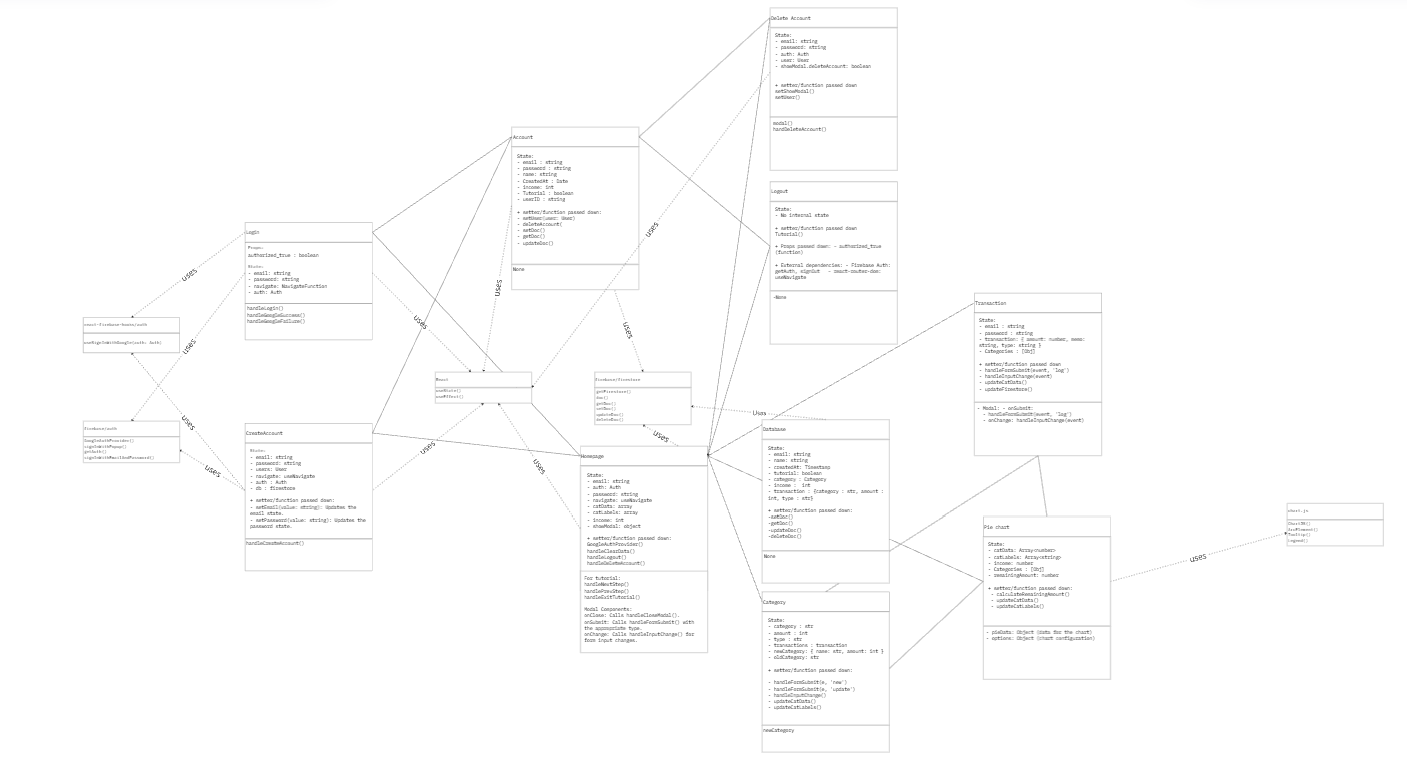
**Design “Class” Diagram**

*<Summary of each part of the design class diagram* [*https://miro.com/app/board/uXjVLEqZCAY=/*](https://miro.com/app/board/uXjVLEqZCAY=/) *if it is decomposed into several parts>*



**Login:**

****

*The diagram above details the Login component, which facilitates user authentication. This component's interactions and dependencies are highlighted through the use relationships, represented by dotted arrows pointing from the using class to the used class.*

**Props and State Management:**

*The Login component maintains state variables for user inputs such as email and password, alongside navigation and authentication utilities (navigate and auth). It also includes a boolean authorized\_true prop to indicate the user's authenticated status. These attributes ensure that the component dynamically manages user input and authentication status throughout the login process.*

**Functionality:**

Key methods in the Login class include:

handleLogin() *for initiating the authentication process.*

handleGoogleSuccess() *and* handleGoogleFailure() *for managing outcomes of Google authentication attempts. These methods interact with external classes and libraries to streamline the authentication process and handle edge cases effectively.*

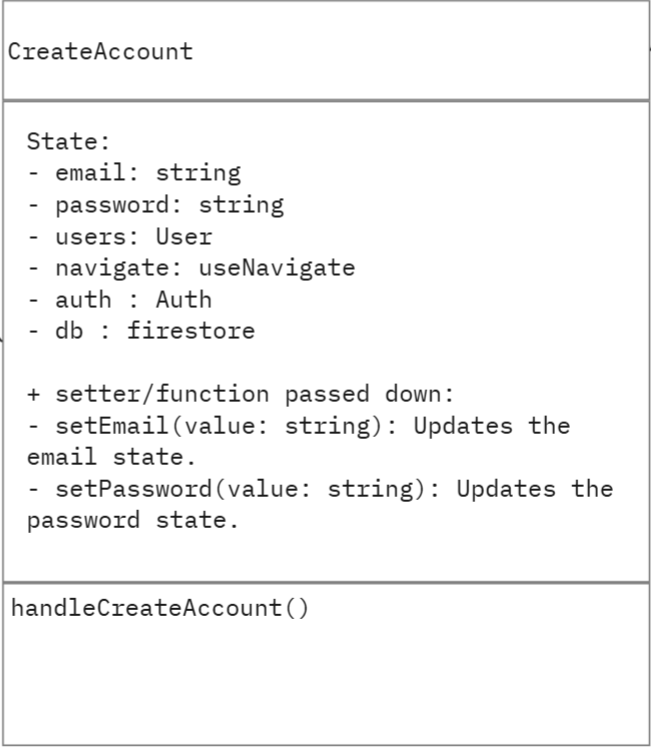
**Dependencies:**

react-firebase-hooks/auth*: Simplifies Firebase authentication, including Google Sign-In, through the useSignInWithGoogle(auth: Auth) function.*

firebase/auth*: Provides foundational authentication functions such as GoogleAuthProvider() and getAuth(), essential for managing user sessions and authentication providers.*

React*: Core React hooks like* useState() *are employed for state management and side-effect handling.*

**CreateAccount:**

****

*The diagram above details the CreateAccount component, which facilitates the creation of new user accounts within the system.*

**State Management:**

*The CreateAccount component maintains several state variables to handle user input and system interactions effectively:*

email *and* password *store the user's input credentials.*

users *tracks user data and is integrated with database operations.*

navigate *and* auth *manage navigation and authentication processes, respectively.*

db *provides access to the Firestore database.*

*Additionally, setter functions like* setEmail(value: string) *and* setPassword(value: string) *are passed down to update the corresponding state variables dynamically.*

**Functionality:**

*The core method in this component,* handleCreateAccount()*, initiates the account creation process by interacting with Firebase authentication and Firestore. This function ensures that new user credentials are securely registered and stored in the database.*

**Dependencies:**

firebase/auth*: Offers methods like* GoogleAuthProvider() *and* getAuth() *to handle authentication securely.*

react-firebase-hooks/auth: *Provides hooks such as useSignInWithGoogle() for integrating Firebase authentication services seamlessly.*

React: *Utilizes React's useState() and useEffect() hooks for managing component states and lifecycle events.*

Firestore Database: *Stores user details post-account creation.*

**Account:**

****

*The diagram above outlines the Account component, which manages the user's account details and provides functionality for handling data associated with the account. This component interacts with several other modules to ensure secure storage and management of user information.*

State Management:

*The Account component maintains state variables essential for account functionality:*

email*,* password*,* name*, and* userID *store key user details.*

createdAt *records the account creation date for tracking purposes.*

income *and* Tutorial *represent user-specific attributes, like finances and doing the tutorial.*

*These state variables ensure that the component can handle personalized user data and adapt its behavior accordingly.*

Functionality:

setUser(user: User)*: Updates the user details in the state.*

deleteAccount()*: Facilitates the removal of the user account from the system.*

setDoc(), getDoc(), and updateDoc()*: Interact with the Firestore database to store, retrieve, and update account-related data.*

**Dependencies:**

firebase/firestore*: Provides methods for database operations (setDoc(), getDoc(), updateDoc()) to ensure account information is securely managed.*

React*: Utilizes hooks for managing state and interacting with other components within the application.*

**Homepage:**

****

*The Homepage class serves as the central hub of the application, providing an interface for users to view and interact with their financial data. It integrates data from categories and transactions while dynamically updating visualizations like the pie chart.*

**State Management:**

Email, password, auth*: Stores the user’s data*

income*: Tracks the user’s declared income for budget calculations.*

catData, catLabels*: Stores the data for all categories to populate visualizations and summaries.*

transactionData*: Consolidates transaction information for display and analysis.*

remainingAmount*: Tracks the user's remaining balance after deducting expenses.*

**Functionality:**

loadData()*: Fetches user data, including categories and transactions, from the database on component initialization.*

updateIncome(newIncome: number): *Adjusts the user’s income and updates associated calculations.*

updateCategoryData(): *Refreshes the data and labels for categories, ensuring the pie chart reflects current information.*

addTransaction(transaction: object): *Adds a new transaction and updates the relevant data.*

deleteTransaction(id: string)*: Removes a transaction and recalculates affected totals and charts.*

**Dependencies:**

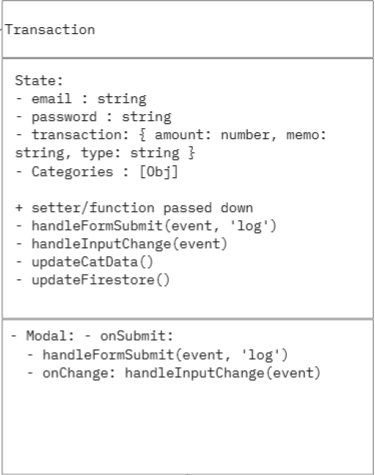
Database*: Supplies the data for income, transactions, and categories, ensuring persistence and accuracy.*

Category*: Provides category details and updates for income and expense tracking.*

Transaction*: Supplies transaction details and facilitates real-time updates*

Pie Chart*: Displays visual representations of financial data, dynamically updated based on state changes.*

**Transaction:**

****

*The Transaction class is responsible for managing individual financial transactions within the application. It handles the recording, categorization, and integration of transactions into the broader financial overview, such as category totals and visualizations.*

**State Management:**

amount*: Represents the monetary value of the transaction.*

transaction*: Consolidates all transaction details for easy access and updates.*

categories: *stores the category it’s associated with*

**Functionality:**

addTransaction(amount: number, type: string, category: string, memo: string): *Records a new transaction with its associated details.*

updateTransaction(id: string, updatedData: object): *Modifies an existing transaction’s details.*

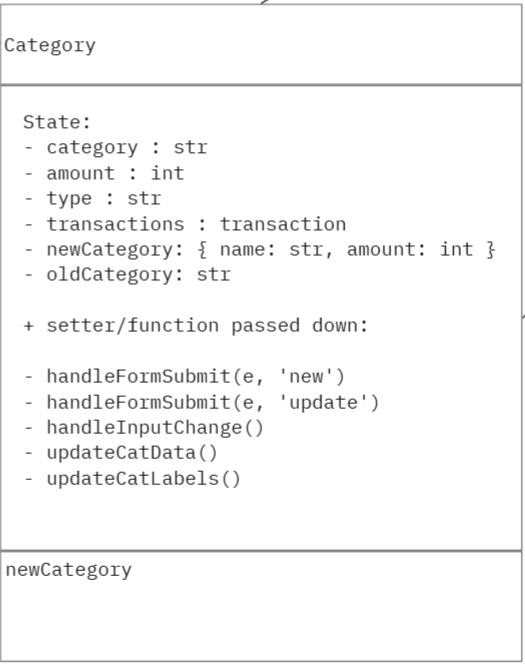
deleteTransaction(id: string): *Removes a transaction by its unique identifier.*

**Dependencies:**

Database: *Handles the persistent storage and retrieval of transaction data.*

Homepage*: Provides an interface for users to add, edit, or delete transactions.*

Pie Chart: *Uses transaction data to update category-specific visualizations dynamically.*

**Category:   
** *The Category class is responsible for managing user-defined financial categories, such as “Groceries” or “Utilities.” It tracks category-specific data, ensuring proper organization and integration with transactions and visualizations like the pie chart.*

**State Management*:***

name*: Stores the name of the category.*

amount*: Tracks the total amount associated with the category.*

type*: Indicates whether the category represents income or an expense.*

catData*: Stores category data for integration with the pie chart.*

**Functionality***:*

addCategory(name: string, type: string): *Creates a new category and assigns its type.*

updateCategory(name: string, amount: number): *Adjusts the total amount associated with a specific category.*

deleteCategory(name: string): *Removes a category and its associated data.*

syncWithDatabase(): *Ensures the category data is synchronized with the database for persistence.*

**Dependencies:**

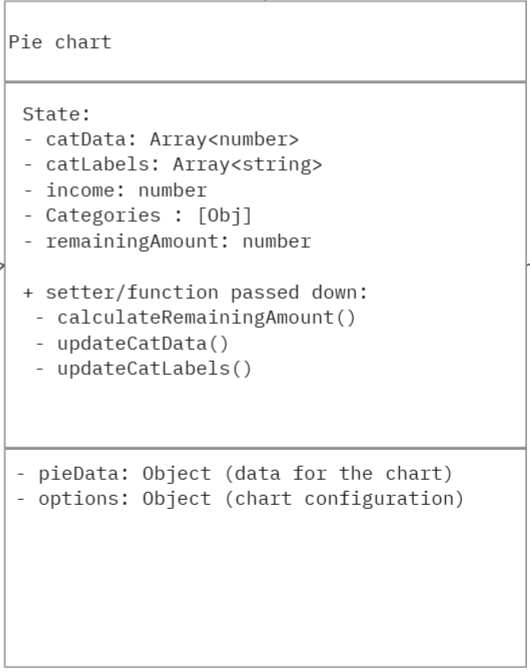
Database*: Handles storage and retrieval of category data to ensure persistence across sessions.*

Transaction*: Provides the amounts that contribute to a category’s total.*

Pie Chart: *Uses catData to update visual representations of the categories.*

Homepage*: Embeds the category management interface and passes functions for creating, updating, and deleting categories.*

**Pie Chart:**

****

*The Pie Chart class is responsible for visually representing the user’s financial data. It integrates with other components, such as Transactions and Categories, to dynamically update and display data in a graphical format. This class ensures that the user can easily understand their financial distribution across categories.*

**State Management:**

catData: *Stores numerical data for each category to be displayed in the chart.*

catLabels*: Stores the labels corresponding to each category in the chart.*

income: *Tracks the user’s stated income as a reference for calculating remaining amounts.*

remainingAmount*: Tracks the user’s income after deducting all expenses.*

**Functionality:**

calculateRemainingAmount()*: Computes the remaining balance after all transactions.*

updateCatData()*: Updates the numerical data for each category to reflect changes.*

updateCatLabels(): *Updates the category labels dynamically as categories are changed.*

**Dependencies:**

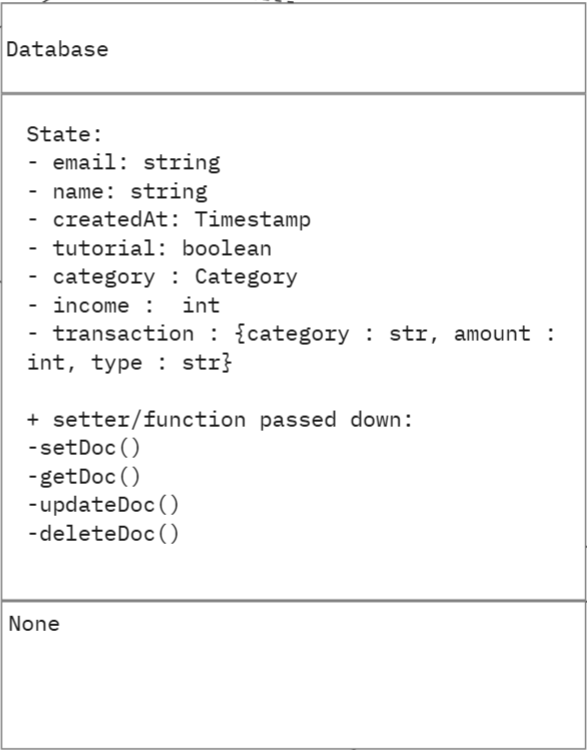
Transaction*: Supplies the financial data for the pie chart, including transaction amounts and their associated categories.*

Category*: Provides the category labels and ensures the data displayed in the chart corresponds to user-defined categories.*

Database*: Acts as the source of truth, storing and retrieving catData, catLabels, income, and remainingAmount to ensure the pie chart reflects accurate and persistent data.*

Charting Library*: Used to render the visual representation of the pie chart.*

Homepage*: Embeds the pie chart and ensures it updates dynamically.*

**Database:  
**

*The Database class manages all data storage and retrieval operations for the application. It interfaces with Firebase Firestore to handle persistent user data, ensuring seamless integration between the frontend and backend. This class centralizes all database operations and supports create, read, update, and delete functionalities.*

**State Management:**

email and name*: Store the user's credentials.*

createdAt: *Stores the timestamp when the account was created.*

tutorial: *Stores whether or not the user has done the tutorial*

category: *Tracks a specific category*

income: *Tracks the users stated income*

transaction: *Tracks each transaction*

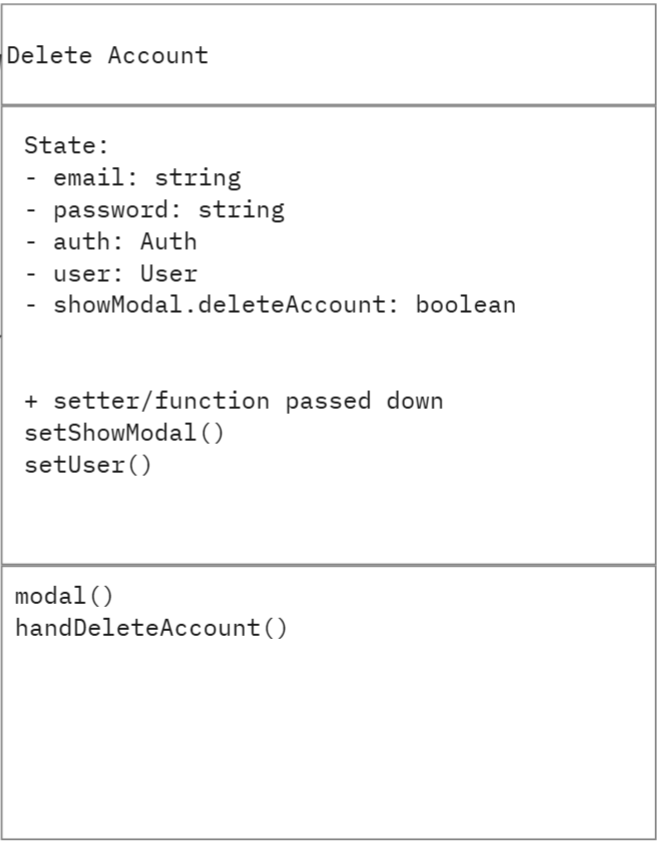
**Functionality**:

setDoc()*: Create a new user, transaction, or category in the db*

getDoc(): *Fetches a piece of information*

updateDoc(): *Changes a piece of information*

deleteDoc(): *Removes a specific piece of information*

**Delete Account:  
**

*The diagram above describes the Delete Account component, which is responsible for managing the functionality and user interface for account deletion. This component ensures that users can securely and intentionally remove their accounts from the system.*

State Management:

email *and* password*: Store the user's credentials required to authenticate the deletion process.*

auth*: Provides the authentication instance necessary to verify user identity.*

user*: Represents the current user object associated with the account to be deleted.*

showModal.deleteAccount*: A boolean flag that determines whether the confirmation modal for account deletion is visible.*

**Functionality:**

setShowModal(): *Toggles the visibility of the deletion confirmation modal.*

setUser(): *Updates the user object in the component state.*

modal(): *Manages the display of the confirmation modal to prevent accidental account deletions.*

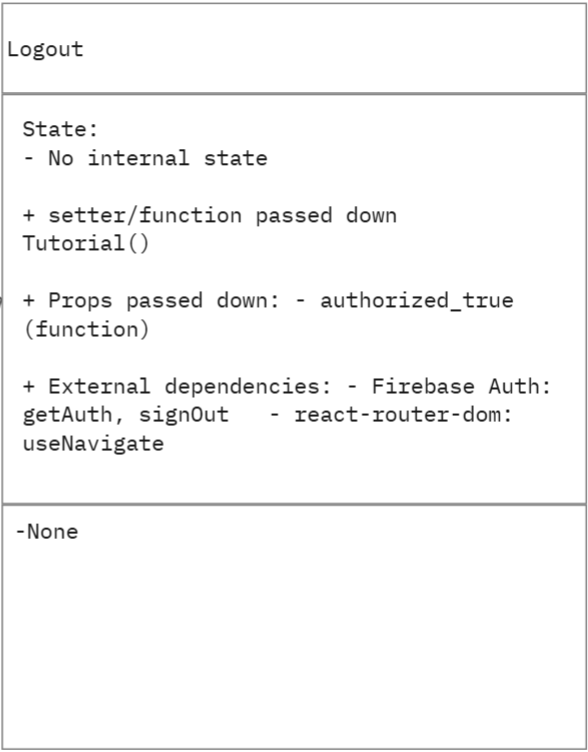
handleDeleteAccount(): *Initiates the account deletion process by interacting with the authentication system and the Firestore database to remove the user's account data securely.*

**Dependencies:**

firebase/auth: *Provides the authentication mechanism necessary for validating user identity during account deletion.*

React: *Utilizes React's state management tools to maintain user input and modal visibility.*

**Logout:**

****

*The diagram above details the Logout component, which is responsible for securely logging users out of the application. This component ensures that user sessions are terminated properly, safeguarding account security and preventing unauthorized access.*

**State Management:**

*The Logout component does not maintain internal state, keeping its functionality lightweight and focused solely on the logout process.*

**Functionality:**

Tutorial(): *A function passed down as a prop, which ensures that any necessary tutorial state is updated after logout.*

**Props Passed Down:**

authorized\_true*: A boolean value passed as a prop to indicate whether the user has been authorized before logging out.*

**Dependencies:**

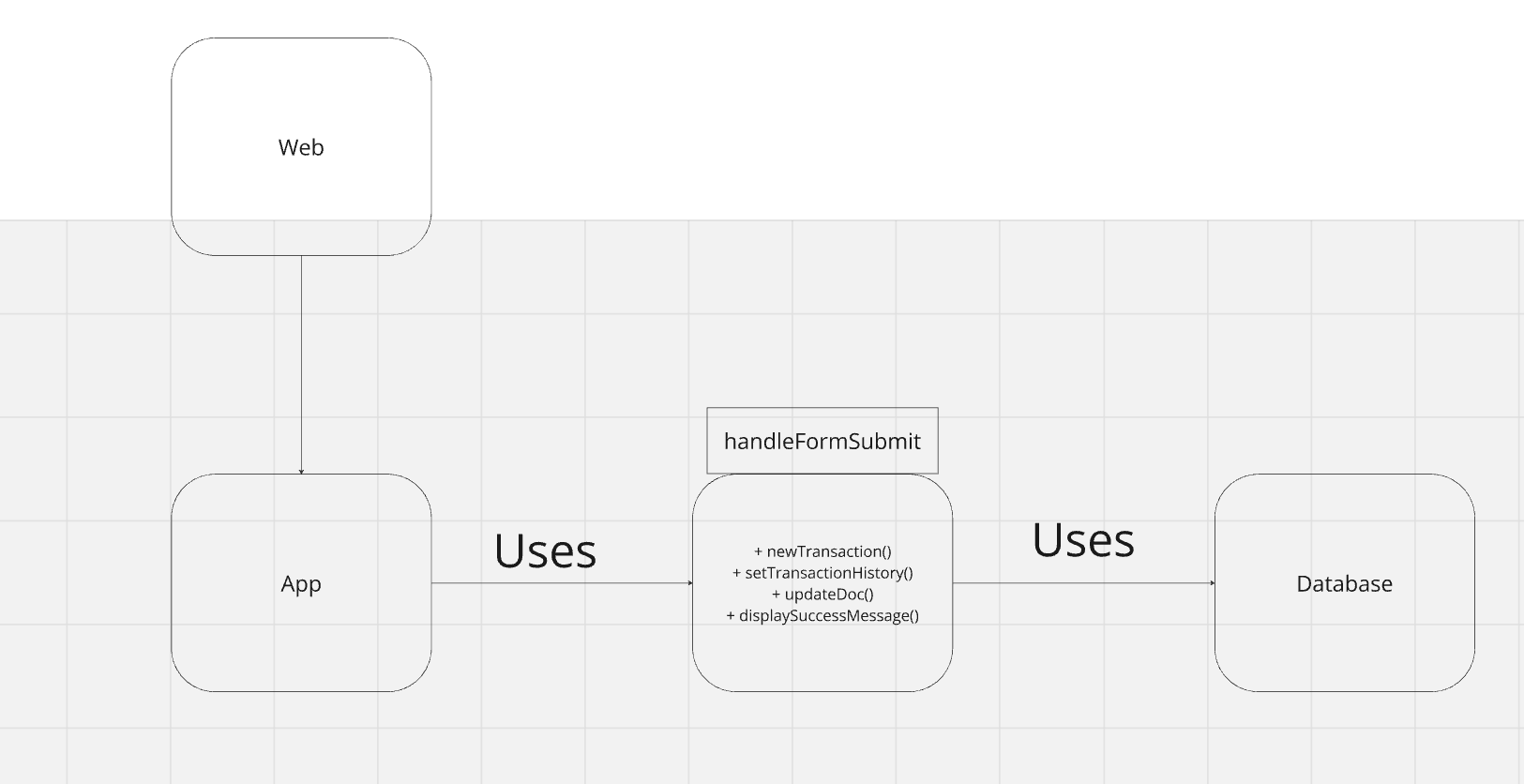
*firebase/auth: Uses getAuth and signOut functions to handle the authentication instance and terminate the user's session securely.*

*react-router-dom: Uses useNavigate to redirect the user to an appropriate page (e.g., login or home) after successfully logging out.*

**Design Patterns**

**Front Controller Design Pattern**

We used the Front Controller Design Pattern on a particular function of our application, handleFormSubmit. In this function, which behavior to execute (log a transaction, update a category, create a new category, or update income) is based on the type of action passed to it. It is the biggest function in the whole app, which can be found on homepage.js. It basically controls all of the inputs within our homepage, and sends it to handleFormSubmit, which will decide what action to take.

**