Materials Collection Dashboard

JIB 4344 - ScrapTech

Cameron Grohler, Hayden Miller, Reginald Smith, Sai Arun Daverpally

Client: Scraplanta – Jonelle Dawkins

Repository: https://github.com/cgrohler3/JIB-4344-ScrapTech

Table of Contents

Table of Figures

Terminology

API: Application Programming Interface. Allows an application to send requests, receiving an information package in return.

Authentication: Verifying the identity of a user, company, etc.

Back-End: The server-side logic and infrastructure powering an application, managing communication with databases, and implementing business logic.

Cloud Firestore (Database): A NoSQL document database provided by Firebase [Google]. Hosted on the cloud, this database provides real-time data storage and synchronization.

Encryption: Security process of encoding data that can only be undone with a secure key.

Expo: A development framework for building and testing React Native apps with preconfigured tools and libraries. Includes Expo-Router, which is used for navigating routing in application.

Firebase (Google Back-End/App Management Software): A platform by Google providing tools for app development, including hosting, authentication and databases.

Front-End: The part of an application the end users directly interact with, including the design, interface, and functionality.

Heat Map: Data visual denoting differences using color gradients. In the following application, denoting zip code donation quantities.

HTTP: A protocol for sending data across the world wide web.

JSON: A file format. Often used for response objects to organize data in key-value pairs.

React Native: A JavaScript framework for building cross-platform mobile apps using a single codebase. Provides multiple libraries to create handle both front-end and back-end features.

UI: User Interface. The visual and interactive elements of an application that users engage with directly.

Introduction

Background

Our application, Material Collections Dashboard, is a mobile application designed to streamline our client's donation management and inventory tracking. Our team, ScrapTech, designed this application to allow our client, Scraplanta, to replace their current pen-paper system, while meeting the standards for accessibility, scalability, and minimalist design. The app allows volunteers to log/track donations with essential details, such as item name, weight, quantity, zip code, and category. For employees, the app provides additional screens for visualizing donation trends through charts and heat maps. The application is built using React Native for the frontend, Firebase/Javascript for the backend, and Firestore (from Firebase) for a database. Our application relies heavily on Firestore because of its vast built-in functionalities and low-cost, scalable database with real-time syncing and offline access. which makes it ideal for handling Scraplanta's growing data needs.

Document Summary

The **System Architecture** section provides insight into our application's architecture using high-level diagrams and informative descriptions. This section helps users understand the interactions between our frontend (React Native) and backend/DB (Firebase/Firestore) technologies, in both static and dynamic flows at a high-level view.

The **Component Design** section provides an in-depth overview of our application by showing the interactions at the component or class level. At this level, we'll demonstrate the communication between individual files and their functionality described in terms of methods and attributes, expanding upon details explored earlier in the architecture section.

The **Data Storage Design** section describes how user data is stored throughout our system. Data from user inputs to the front-end of the application are transferred into the NoSQL cloud database Firestore, hosted on Google Cloud. Recorded donations are stored as distinct documents inside collections, which are retrieved for operations across the application.

The **UI Design** section shows visual and interactive components like layouts, user flows, navigation and accessibility for an intuitive user interface. Each of the UI screens are also evaluated using Nielsen's design heuristic, which provides some insight into many of the design decisions made for the app's interfaces.

System Architecture

Introduction

Our mobile application is designed for inventory management and intended to run on Android devices, specifically Android Tablets. Although we're developing a mobile application, our application follows a layered architecture, where we're breaking the functionality into the front-end layer (React-Native), backend layer (Firebase), and data layer (Firestore). Our rationale for the choice of each layer is given below:

- React-Native is optimal for this application because of its ease of use, cost-effectiveness in access to multiple libraries, and advantage in cross-platform development. These advantages were significant given the need for our application's visual components to be built, updated, and maintained easily. In addition, React Native can integrate with any backend technology, which ensures our application is open to any potential changes in the tech stack.
- Firebase is optimal for this application because it has a built-in backend and data functionality. This allows developers to control user access in the application by handling backend communication (with frontend & database) in one location. In addition, Firebase's cloud storage allows for high scalability and robust storage.

This choice of architecture was essential given our requirements because we wanted our application to be scalable without changing the entire codebase. Also, such a separation of functionality can help other developers upgrade the application for future releases after handing it over to our client. Given our application was restricted as an internal tool for our client, our goal of creating a streamlined application that could handle large amounts of data was fulfilled by our architectural choices while maintaining security and upholding standards.

In terms of security, the primary security requirement within our application relates to the storage and retrieval of login information. To adhere to this requirement, we chose Firebase's built-in authentication to ensure that login information was stored and encrypted. Even database owners won't be able to access the user's secret information, only metadata about the user, such as email or username. In addition, Firestore's authentication allows for special rules to ensure only authorized users have access to protected routes, while basic users can only access unprotected routes. This choice ensures there's little room for data leakage or vulnerabilities if the client wants to upgrade the application.

Below, we'll provide a high-level overview of our architecture using static and dynamic architecture diagrams. The architecture diagrams below give insight into how our application engages with the user to save, record, and display donation data per our client's requirements.

Static System Architecture

Diagram

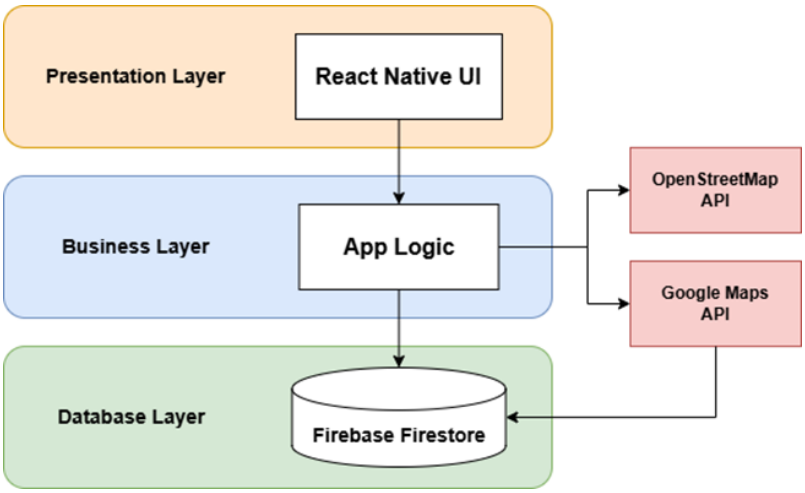

Figure #1: Static System Architecture

Description

For our static system architecture, we followed a layered-architecture design to display the separation of application logic between different layers. This design was motived due to our client's volunteers being of various ages with varying degrees of technology experience. Our goal was to streamline the design for quick communication between UI, the app logic, and the database, which prevents functionality from being congested/dependent on one component. In addition, this design helps us outline the major components of our application and the interactions between them. Our architecture has three main components/layers: **Presentation** (frontend), **Business** (backend), and **Database**.

The presentation layer consists of the React Native UI, which refers to all the visual screens that our client's employees and volunteers will use to access different functions, such as logging donations and viewing heatmap of donations. The presentation layer communicates with the business layer, which houses the internal app logic, which facilitates the communication between the presentation and database layers.

The business layer handles all data processing within the application and writes to and retrieves from the database directly. The database layer consists of the Firebase Firestore, which is where our application stores all the donation data in the form of collections, which store key-documents values. In addition, our app logic component is dependent on two APIs: OpenStreetMap, which is used for latitude/longitude to zipcode conversion, and Google Maps, which is embedded directly into our application. Lastly, Google Maps API is dependent on Firebase Firestore for retrieving data to support heap map visualization.

Dynamic System Architecture

Diagram

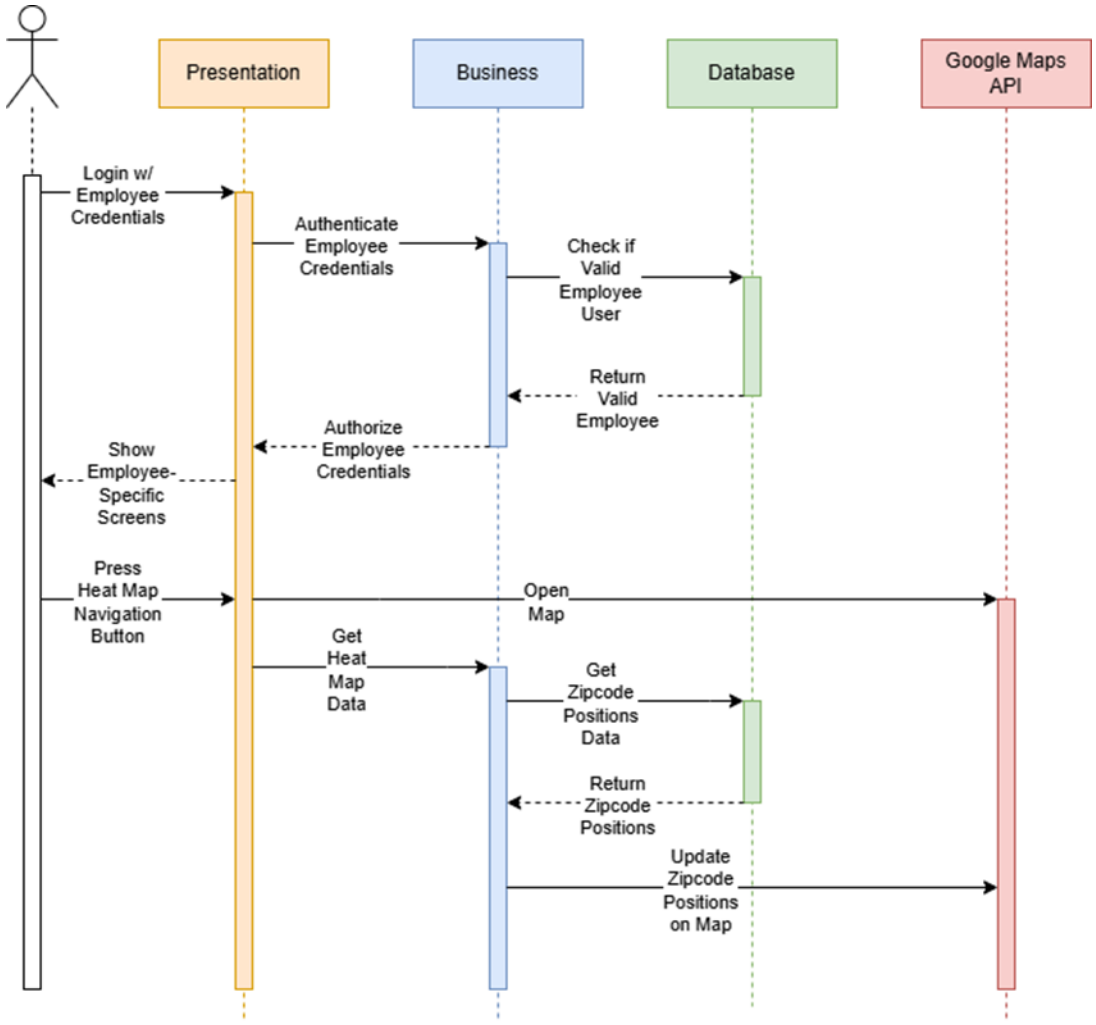

Figure #2: Dynamic System Architecture

Description

For our dynamic system architecture diagram, we followed a system-sequence diagram (SSD) to show the communication between the different layers during one specific scenario, which, in this case, relates to the login of an employee user to view the Heap Map screen.

Initially, upon opening the application, the user is presented with a login screen to login using volunteer or employee credentials. In this case, our user logins in with employee credentials, which upon validation from the business and database layers prompts the presentation layer to return employee-specific screens, which includes the Heat Map screen. The database stores the list of current employes, and the app logic component uses the “employees” collection to

validate the employee user. These screens, however, are hidden when a volunteer user is validated by the business layer.

When an employee user clicks the Heap Map button on the navigation bar, the user is immediately prompted with Google Maps. This Google Maps API is not directly controlled by our application, instead we utilize an external React Native library that controls the API. Our access to his library allows us to send data to this API directly. However, this map will persist until the user clicks on a different screen. Upon loading the map, the Heap Map screen will internally populate the map with new zip code positions by obtaining such data from the database layer. This communication between the business and database layers ensures that the Heat Map is always populated with up-to-date data.

We chose this specific scenario because it showcases interactions between most of the components in our system architecture. These interactions are important to visualize to help our client and other developers understand the architecture at a deeper level and to provide a visualization of the entire flow for this scenario. Although this diagram shows gaps between requests and responses, the entire flow completes/occurs very fast with minimal delay, ensuring no major noticeable difference to the user in the React Native UI.

Component Design

Introduction

To provide a more detailed view of our design, we showcase static and dynamic component diagrams, which help to view interactions between the low-level components in our application. Specifically, for the static diagram, we chose the Component Diagram because our application consists of multiple functionalities and this diagram helps to aggregate them into individual, high-level components. In addition, our application doesn't use objects, rather the presentation layer uses various files (.js & .jsx) as modular JS components to guide communication for functionality, which translates naturally to a component design. For the dynamic diagram, we chose the System Sequence Diagram to highlight the process flow between different files for a non-trivial scenario. This diagram helps provide more granularity on the interactions by exploring the individual files involved in fulfilling a specific process. This diagram is optimal because it helps other developers understand the function of various files, which can help tremendously during development/update of existing functionality.

Static

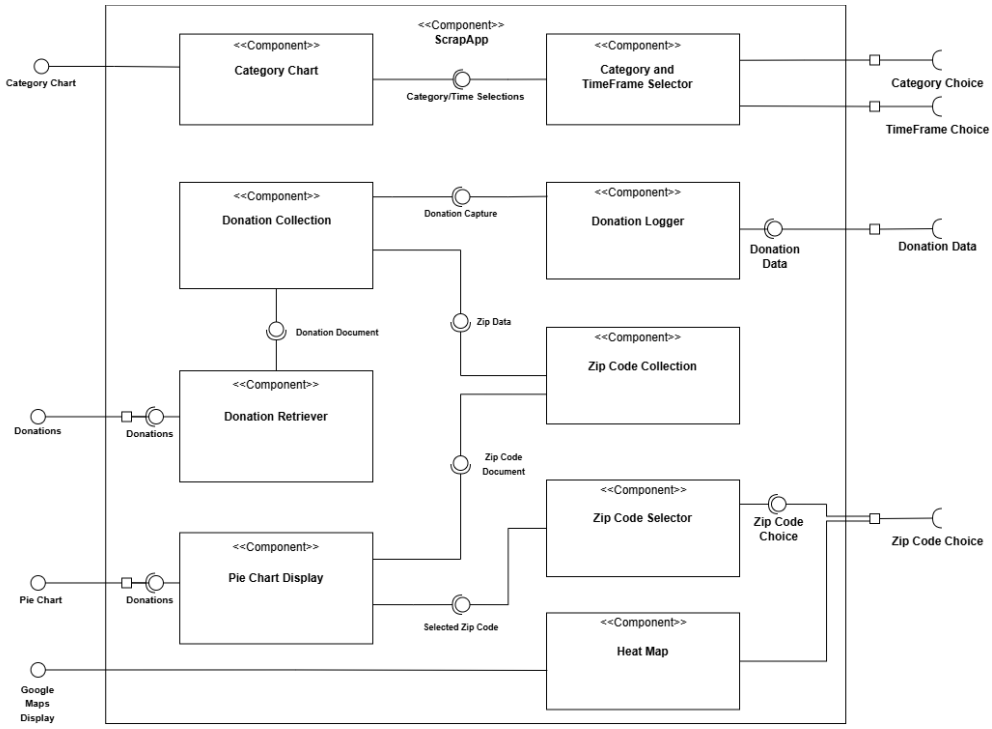

Figure #3 – Static Component Design Diagram

Our static diagram depicts the relationships between different components of our application. We divided our application into five parts: the donation, the zip code of the donation, the pie chart display, the heat map, and the category chart. The **Pie Chart Display** pulls interfaces from the **Zip Code Selector** and **Zip Code Collection** to retrieve the data needed for volunteers. This way, the donation data is grouped by location and can be represented in terms of the donor's zip code. Donations are tracked by the **Donation Retriever**, which draws from the **Donation**

Collection to display any data the user wants to see. The **Donation Retriever** can display any donations saved in the database, so that users can confirm successful donations and review past data. The **Donation Collection** draws from the **Donation Logger**, which is responsible for ensuring donations are stored in the database properly. These three components work together to make sure donation data is entered, stored, and retrieved properly. The zip code data from those donations is also moved to **Zip Code Collection** to be included in the data visualization. The **Heat Map** draws from the **Zip Code Selection** to populate the points of interest on the map display to provide insight on the location in which donations arrive from. The **Category Chart** draws from the **Category Selection** and the **TimeFrame Collection** to populate the bar chart and display category donations over various timelines.

Dynamic

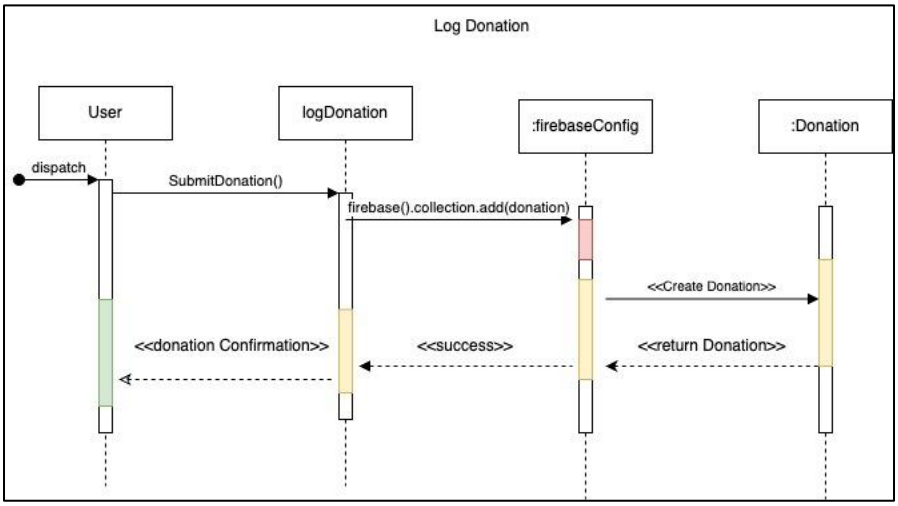

Figure #4 - Dynamic Sequence Diagram

The following sequence diagram illustrates the runtime behavior of the system when a user logs a donation in the application. The process begins when the user interacts with the LogDonationScreen, entering item details about the donation. When submitting the screen constructs a donation object and initiates a call to the firebaseConfig service, which manages communication with Firestore.

The firebaseConfig forwards the request to Firestore, where the donation data is stored as a new Donation document in the donations collection. The data is then successfully saved. The logDonation screen then displays a confirmation message to the user which completes the interaction.

This dynamic behavior aligns with the system's static architecture by showing how low-level components like screen modules and backend services collaborate to complete a user action. The creation of the Donation object is represented to maintain a conceptual view between the static class diagram and the runtime scenario.

Data Storage Design

Introduction

This section provides a detailed description of our database and explores the format of our databasing using UML diagrams. This section also provides an overview of our project's file structure, and the interaction between individual files. Because our project is built using Expo, our file structure follows the folder/file structure of expo, where the codebase is separated into multiple folders, and across multiple files. Our database structure involves non-tabular structure due to the integration of Firebase's NoSQL database.

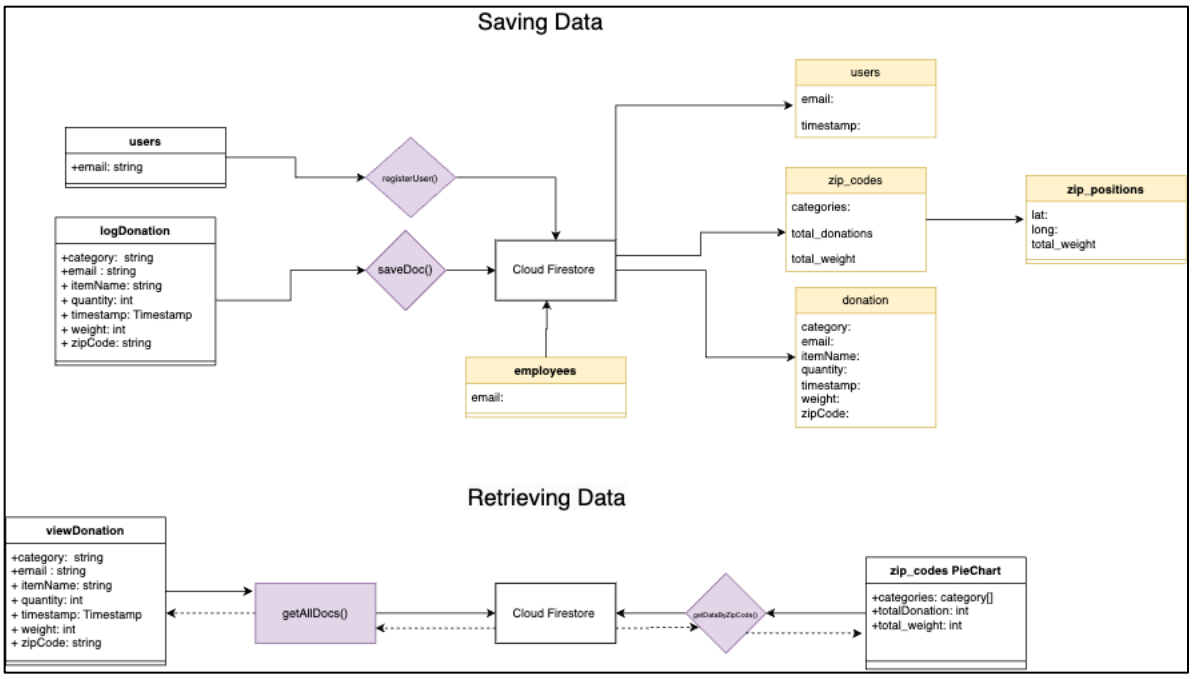

Figure #5 - Data Design Diagram

Using Firebase creates a simple flow for all of the data in the application. We store all our data in collections, which are yellow in the diagram. A user first logs a donation (logDonation) entering all the required information. A saveDoc() function then saves the data in Firestore and you're able to see how the donation is stored using the arrows pointing towards the collections. Collections are stored in JSON format. Employees are entered manually into the database and users are saved to the database upon registering in application. The zip_codes collection adds a new zip code to the collection whenever there is a donation with a new zip code that hasn't been entered before.

When retrieving data, we make calls like getAllDocs() and getDataByZipCode() that queries the database stored from logging donations and returns information, which is the used by the app.

File Use

This project was implemented using Expo and React Native, which have built-in specifications for the file structure to use. All major files are stored in folders: app, assets, and helpers. There are some hidden folders, which are only accessible for developers: lib, which stores the API key for our database, and node modules, which stores all the packages used in our project. Our routing is performed using Expo Router, which used folder structure to create routes. In our project, our routes are specified in "(auth)" folder, which can only be accessed if successful login. Using JS, we export values and functions within files, which are used in other files to run the application smoothly. For example, our secret file, storing API key, exports "db," which is used by other files to connect to the Firebase DB.

Data Exchange

To facilitate communication between client-side and server-side, we utilized HTTP protocols using CRUD requests to the database. These requests allow for the sending, retrieving, and updating of data in our database. This exchange is restricted to specific hosts, or Scraplanta's store locations, using configurations in Firebase.

Data Security

Our application handles personal data for Scraplanta's donors, so protecting the data it handles is necessary at all levels. Scraplanta does not record financial data from donors such as credit card or bank information, but personally identifying information and location data is kept for review and tax return purposes. Advanced encryption is not necessary because most of the donation data is trivial, but for personal identification information, we have database encryption that allows us to protect users and donors. Our application ensures data security by moving and accessing data in as few steps as possible. Data is handled either by storing it or retrieving it in our Firestore database, so that none of it is stored locally without encryption. Firebase automatically encrypts all our data to ensure it cannot be illegitimately accessed. Storing the data in Google's remote database ensures it is well-encrypted and protected by trusted software.

Users who have proper access to the application are required to log in via Firestore's authentication function, so their data is also stored in our designated database. Users without a stored login cannot access the app or data at all. In addition, employees have separate permissions to volunteers, so users can only access data visualization if they have employee-level access. Even those with access to the database cannot see a user's password, so account data is kept private. Emails are public to Scraplanta's employee users, which is not a security concern because Scraplanta already has domain-specific emails their employees use. Login is required to view or log donations through our application, so there is little concern about illegitimate access. The only level of our application concerned with data storage is Firestore, ensuring data is not accessible through multiple routes or susceptible to leakage.

UI Design

Introduction

This section will walk through the key UI screens in the Scraplanta application. The design focuses on ease of use for both volunteers and staff members by having clear data input fields and visual data summaries. The app uses a universal navigation bar that allows users to access any screen at any time, improving usability and preventing the user from getting lost. The screens are minimalistic and centralized, displaying only necessary information related to what the user may be trying to do – access or edit donations, view data, and so on.

The screens below correspond to the major tasks that the user will complete while using the app. Note the icons on the navigation bar highlighted in blue, which correspond to the screen being displayed. The logout button, present on all screens, is highlighted in blue as well for visual distinction. All opaque green buttons mark system functions, such as saving donations and updating the database. Clear buttons represent user-side functions, mainly clearing out forms if data is entered in error. Titles in the top left corner of each screen alert the user as to which screen they are currently on.

Login Screen

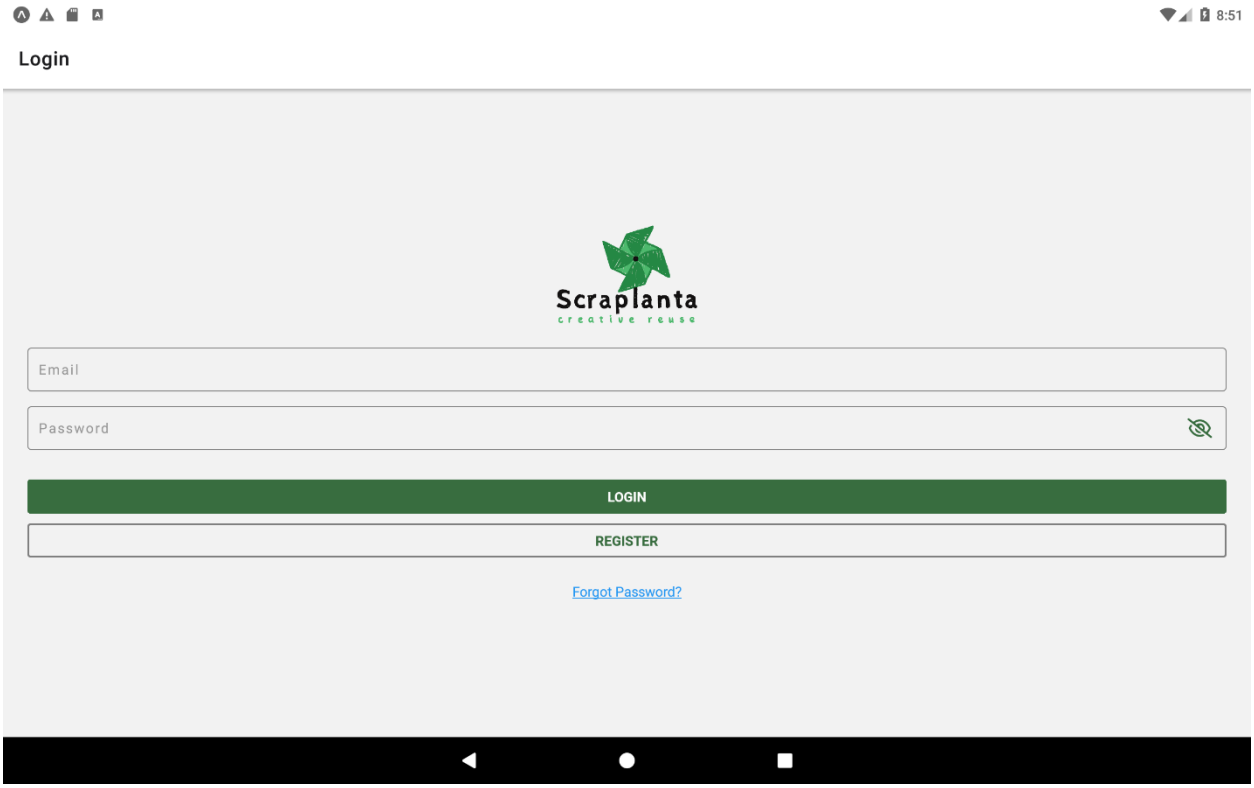

Figure #6 - Login Screen

The login screen is the entry point into the application. This screen is designed to authenticate users and ensure appropriate access based on their role in the organization of either a volunteer or official staff member. Volunteers are only allowed access to log and view donations, while staff members have full access to all functionality within the application like logging and viewing donations, accessing zip code and material data, and full access to the heat map.

Heuristics:

- User Control and Freedom – The login screen upholds this heuristic by allowing users complete freedom in how they want to perform authentication. Users can either register (if not already), or login using created credentials. If users enter the wrong authentication window, then they can easily switch to the other window with the touch of a button.
- Consistency and Standards – The login screen follows industry design standards for authentication screens, where users can type passwords that are hidden, unless specifically viewed by the user. Input boxes are labels in simple language, making it easier for users to recognize what information to fill.
- Help Users Recognize, Diagnose, and Recover from Errors – If users try to authenticate with incorrect credentials, or register with registered credentials then they'd be prompted with errors messages that notify them of the error and provide solution to overcome the error.

Log Donation

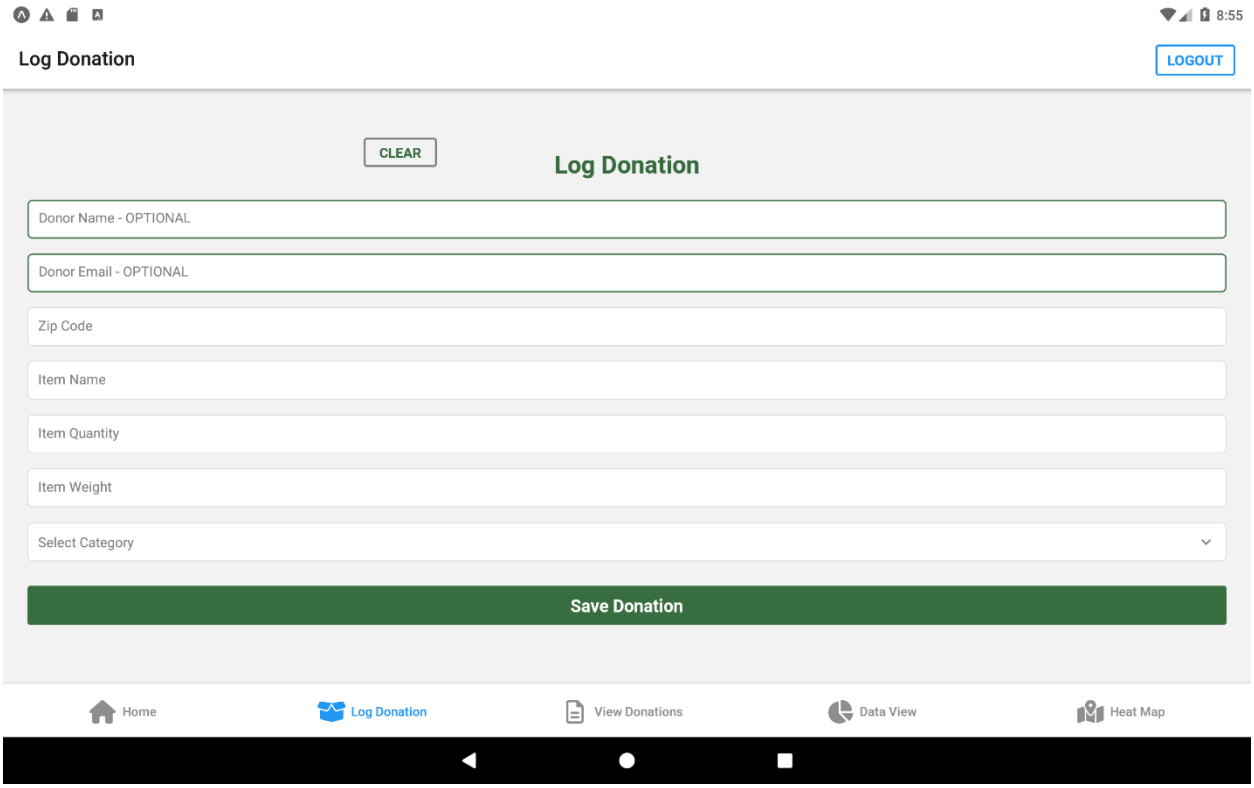

Figure #7 - Log Donation Screen

The Log Donation Screen is the core interface for recording incoming donations. This screen allows a smooth data entry process for volunteers and staff, making it quick and easy to log every donation. Users are prompted to insert essential information about the donor and the materials being donated through clearly structured fields. This screen plays a crucial role in maintaining the accuracy and consistency of the donation database. By making sure every donation has this standardized format, we ensure a reliable process for donations and allows Scraplanta to have accurate analyses and reporting.

Heuristics:

- Visibility of System Status – The log donation screen provides the users with popup alerts to notify them about the status of their actions. These alerts occur in response to user actions on the screen, like clicking the save button or entering info into the input boxes and help notify the user of the system status.
- Error Prevention – This screen also prevents error-prone conditions, like preventing users from entering text into numeric inputs. For example, if a user tries to save without entering all the required information, then the system will alert the user of this status. These alerts and safeguards ensure users don't make mistakes, even by accident, or fix their mistakes if it's expected and accounted for in the system.
- Aesthetic and Minimalistic Design – This screen uses simple input boxes to focus user attention on one task only: logging a donation with the correct information. By keeping the design minimalistic, we allow users to be focused on the task at hand, and easy flow during task completion.

View Donation

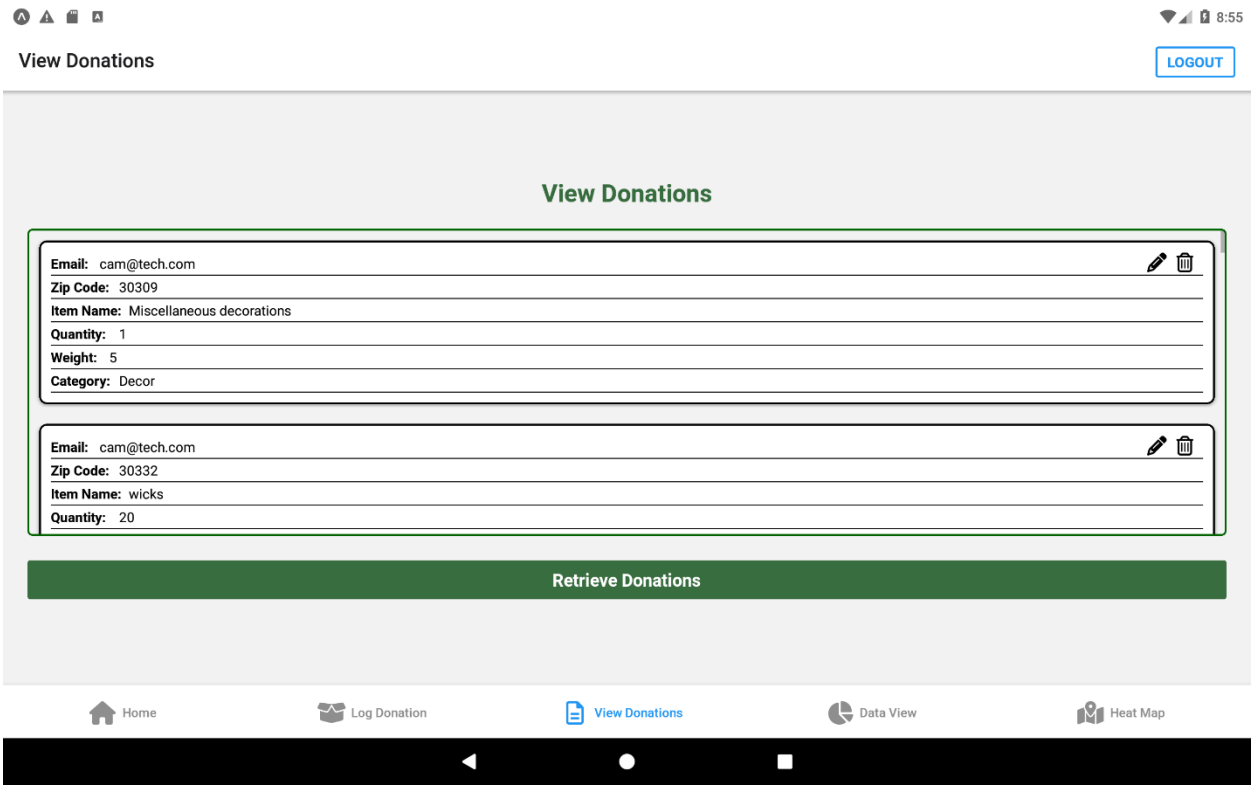

Figure #8 - View Donation Screen

After logging donations, users can visit this screen to retrieve and view recently submitted entries. By clicking the "Retrieve Donations" button, a list of donations appears and displays the essential information for donations like email, zip code, item name, weight, and category. This screen ultimately allows for data verification.

Heuristics:

- Aesthetic and Minimalist Design – This screen provides no irrelevant data. A title, button, and list detailing the data fields that a user expects to retrieve are displayed. With no extra options or distracting elements, users are prevented from accomplishing anything except for the task the screen is designed to accomplish.
- Recognition Rather than Recall – This screen includes the names of each data field, preventing users from needing to recall what they have entered upon logging a donation. It ensures that a number like 'quantity' is not to be confused with 'weight' or 'zip code.' No guess work is required by the users.
- Visibility of System Status – This screen updates immediately upon selecting the option to retrieve donations. Choosing not to press the button leaves the donations field empty, reflecting the users' lack of action. With only the one button, providing input and seeing the change in state displays to the user that their actions have successfully influenced the application.

Zip Code Pie Chart

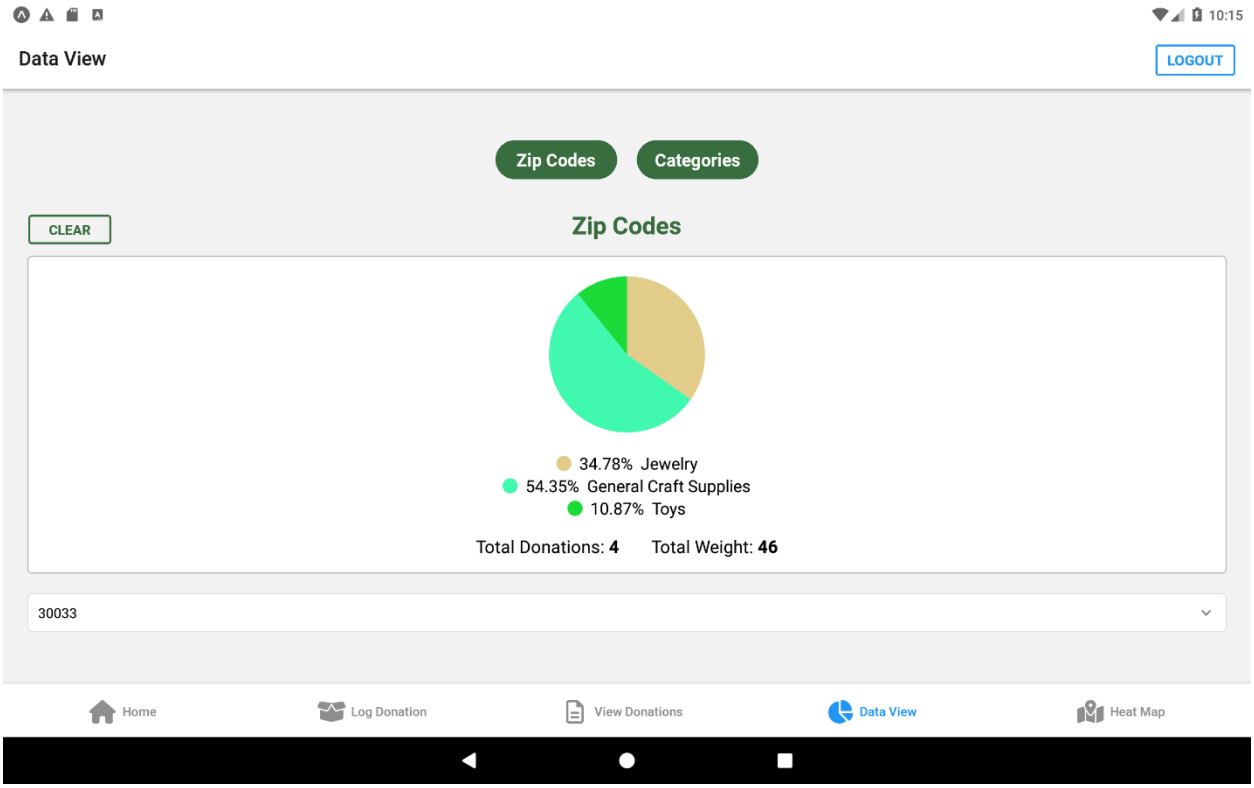

Figure #9 - Zip Code Pie Chart Screen

This screen allows users to select a specific zip code to analyze donation data associated with that specific zip code. Once selected, a pie chart dynamically displays the percentage breakdown of donation categories within that zip code. Other metrics include the total number of donations and total donation weight to give users a more comprehensive view of that area's activity.

Heuristics:

- Consistency and Standards – This screen maintains external consistency with pie chart displays used in research papers as well as other applications, having a key describing category details and distinct colors that allow for analysis of the data.
- Visibility of System Status – This screen updates the title as input is given to select a zip code and continues to do so over multiple selections. This ensures that choosing different options from the dropdown menu indicates to the user that their choices are reflected in the current pie chart display.
- Aesthetic and Minimalist Design – This screen prioritizes the pie chart, having it dominate the screen since it is the primary element of the screen. The option to toggle between zip codes and materials is stored above the chart and title, keeping it separate from the data visualization provided. Since the totals do not include visuals, they are relegated to text summaries that do not distract from the colorfulness of the chart.

Donation Materials Bar Graph

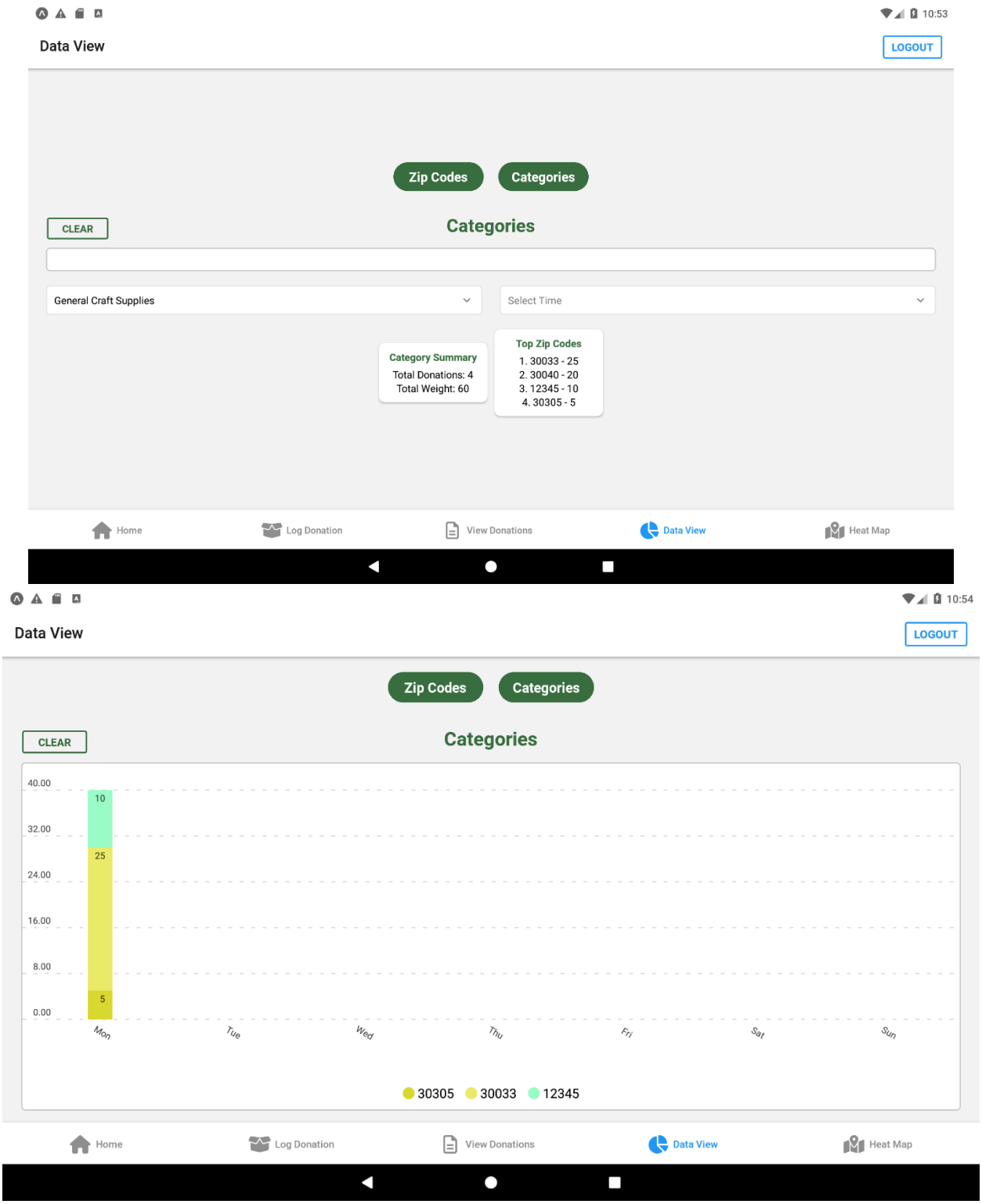

Figure #10 - Materials Screen

This screen focuses on material-specific data across all zip codes. A dynamic bar chart shows total donation weight by material category which gives the users insight into which materials are

most commonly donated. Also, below the chart, a ranked list shows the top contributing zip codes by total weight which shows the high donation areas.

Heuristics:

- Visibility of System Status – The material graph automatically reacts to new inputs by the user, clearly showing that the data displayed depends on the user's selection. The bar graph is clearly labeled, so it is obvious what is being displayed and how it correlates to the user's selection. The zip code breakdown at the bottom shows the user exactly what data they are seeing and where it came from.
- Aesthetic and Minimalist Design – The screen only has a few elements, which are either labeled buttons or the bar graph itself. There are no confusing or distracting elements to take away from the screen's goal, and the sparsity of information makes it easy for the user to focus on the information they need. Relevant information is the focus of the screen.
- User Control and Freedom – The navigators on this screen are easy to access and visible at all times. If the user reaches this screen by accident, or if they need to revert back to the pie chart display, clearly labeled navigation buttons allow the user to backtrack and switch between views. A refresh button is provided, and the drop-down menu is easily found if the user makes a choice on accident and needs to undo it.

Interactive Heat Map

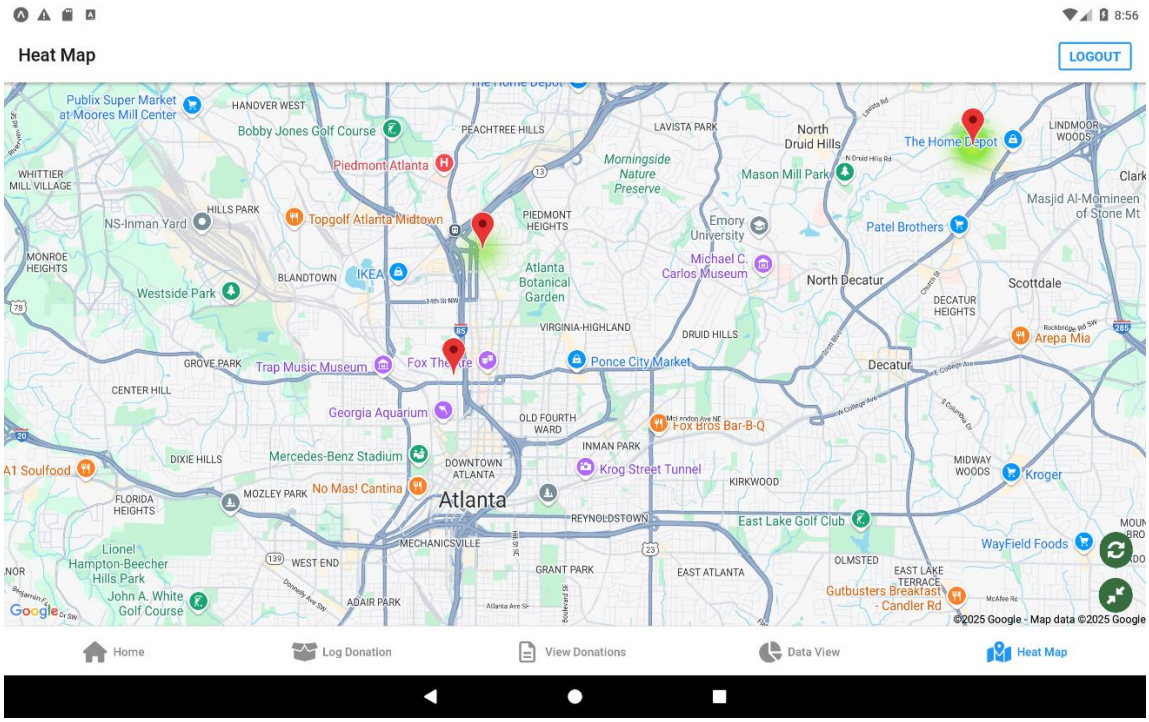

Figure #11 - Heat Map

The heat map provides a visual overview of where donations are coming from across City, State, and Country scopes. The intensity of the heat signature visually represents the weight of donations from a zip code. Brighter areas show heavier donation volumes. The screen shows a spatial view of donation activity and highlights high and low activity areas on the map.

Heuristics:

- Recognition Rather Than Recall – The navigation buttons on the right are clearly marked and stand out from the map. Rather than full labels, the buttons make use of simplified icons that tell the user what they do in a succinct manner using images that are universally recognizable.
- Match Between System and Real World – The heatmap mimics the conventions of a real map or online navigation tool. Hotspots and markers are used to draw the eye to relevant or important locations. The map also uses zoom and scroll features that allow users to examine the map in the same way a real map would be used. The map's appearance mimics the location symbols used in the real world.
- Aesthetic and Minimalist Design – The map avoids clutter and overload of information by using a simple interface. The only interactive buttons are small and off to the side, easily accessible but not detracting from the main goal of the map display. The map is easily visible and is not covered by information or text – rather, heat spots and markers are used to alert the user to points of interest. The user can still navigate and easily read the map with these markers included, so that one piece of information does not dominate the screen.

Appendix

OpenStreetMap API

Resource URL
https://www.openstreetmap.org/

Resource Information
Response Format: JSON
Requires Authentication: No
Rate Limited: Max of 1 request/sec

Parameters
Postal Code

Example Request
https://nominatim.openstreetmap.org/search?postalcode=30332&country=US&format=json

Example Response

[{"place_id":353640239,
"licence":"Data © OpenStreetMap contributors, ODbL 1.0. http://osm.org/copyright",
"lat":"33.773023389473686",
"lon":"-84.39318890789474",
"class":"place",
"type":"postcode",
"place_rank":21,
"importance":0.12000999999999995,
"addresstype":"postcode",
"name":"30332",
"display_name":"30332, Atlanta, Fulton County, Georgia, United States",
"boundingbox":["33.7230234","33.8230234","-84.4431889","-84.3431889"]}]

Response Fields
The only response fields relevant to this application are that of “lat” and “lon”. These fields denote the latitude and longitude of the requested zip code to be used for points of interest on the heat map.

Team Contact

Name	Role/Responsibility	Contact
Cameron Grohler	Team Lead / Database Design- Maintained communications with client to meet project expectations. Designed and implemented the organization of data in Google's cloud Firestore to optimize read and write access to database.	217-323-4955 cgrohler3@gatech.edu
Reginald Malik Smith	Front end developer – Developed the dynamic zip codes pie chart and dynamic top zip code summary data while also making several UI improvements to make the application more visually appealing.	706-619-7943 reginaldsmith611@outlook.com
Hayden Miller	Frontend Developer - Developed the dynamic heatmap and provided UI for data visualization, app navigation, and user interactivity. Responsible for quality-of-life updates and ensuring a functional, polished final product.	404-550-3135 haydenmiller242@gmail.com
Sai Arun Daverpally	Backend Developer – Developed the backend logic for a majority of the screens and created supplemented functions/methods to allow for live-updates on screen. Implemented connections to the database to allow for dynamic content, saving, and retrieval on the fly.	470-628-3309 dav.saiarun@gmail.com