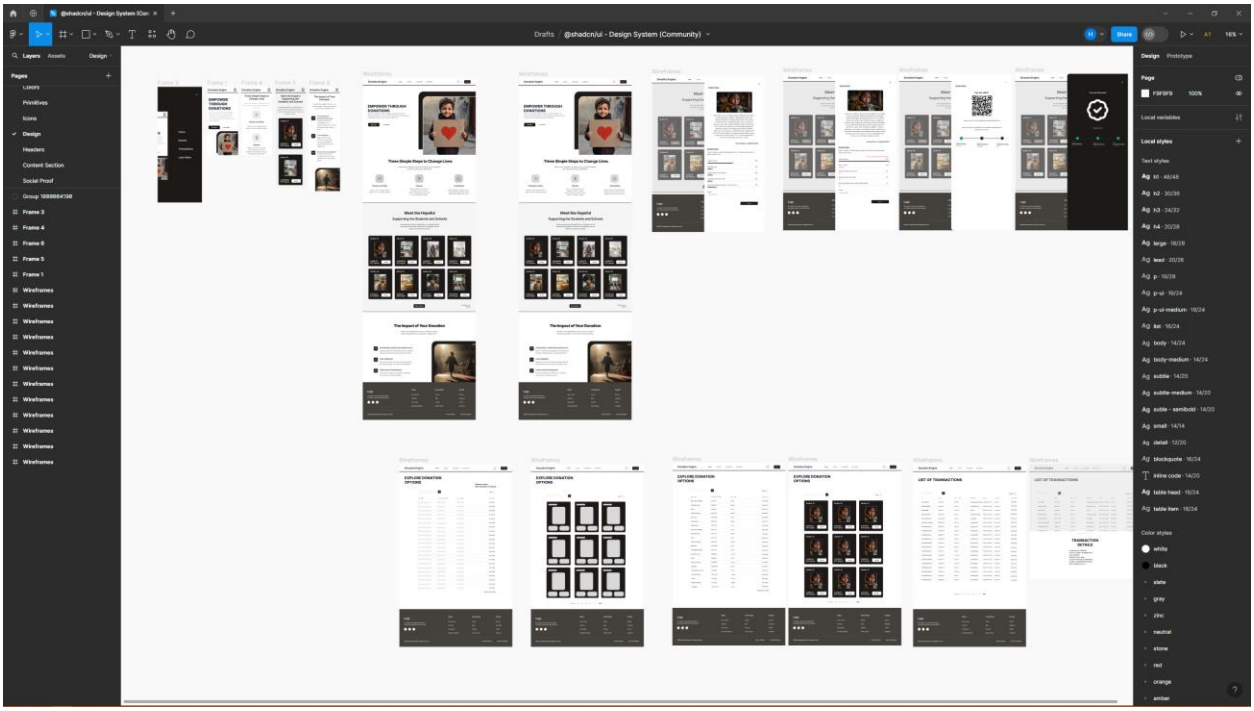
Stellenbosch hackathon Support Documents:

Overview:
This project is built for the hackathon in Stellenbosch South Africa. Requiring teams to build a donation engine that supports bitcoin integration on the internet computer while being architecturally sound for future cryptocurrency support.

The current technology stack used is Motoko for the backend, Svelte for the front-end, Tailwind for styling to write fast inline styles, and Shadcn-Svelte for prebuilt lightweight customizable components. The project is bundled using Vite which optimizes the final build for both svelte and tailwind removing any unnecessary unused code/imports and CSS styles while efficiently bundling them to server to the end users. Additionally, the website has progressive web app (PWA) enabled. This enables the users to “download” the website mimicking a native mobile application. (Note* This feature might not be available on iOS devices as PWA’s are being restricted by apple at the time of writing this). Additionally, a service worker is coded that sits between the frontend and the backend enabling the caching of assets so that the server is not constantly responding with images or fonts for every request reducing the overall load on the server. Lastly, if a new version of the application is built the service worker will invalidate the cache so that users get the latest changes.

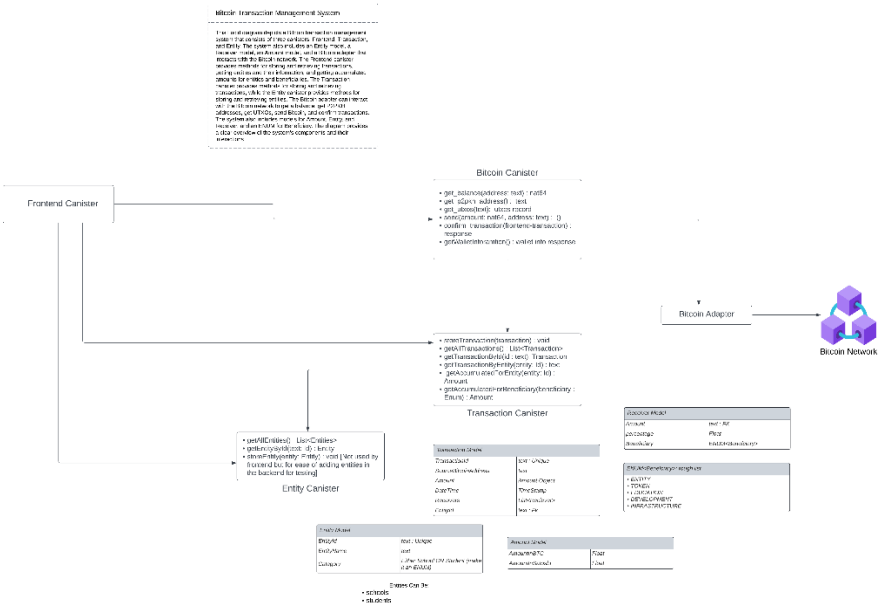
Mockups:
Mockups are done using Figma. The mockups are used as guidelines for building the frontend. Although they are not the final design as requirements are ever changing the mockups must be changed accordingly. Full view pictures can be viewed in the design documents in the GitHub repository.



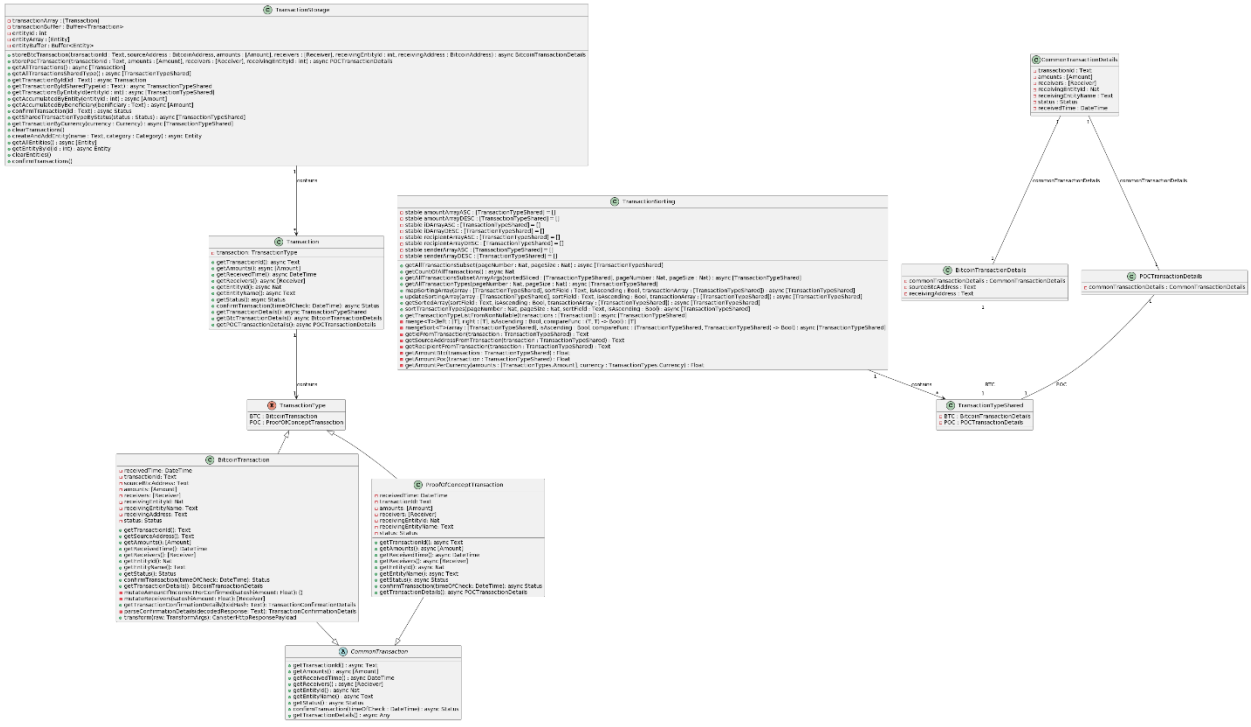
Backend Architectural Design:

Full view of the images can be found in the design documents folder of the GitHub repository.

Initial Design:



Final design:



The backend contains two main canisters. The bitcoin canister that interacts with the internet computer native bitcoin API calls, and the transaction canister which contains two sub canisters one for storage of transactions and entities and another for sorting the transactions. Lastly, a commons folder that contains the types needed for all canisters. Divided into types for bitcoin, entities, transactions, http requests, and JSON parsing.

The bitcoin canister implements the native bitcoin API functionality from the internet computer. Including functionalities such as retrieving the p2pkh address, getting the wallet balance, and unspent transactions. No data processing is done using this canister. It is used as a medium for information calls for the transaction canister and the frontend.

The backendTransaction canister contains the transactionStorage and transactionSorting functionality. The storage has an abstract class called Common transaction which can be used as a basis for multiple cryptocurrencies. The abstract class is extended to be used in BitcoinTransaction and ProofOfConceptTransaction which are of type TransactionType. The TransactionType can be used in functions to specify or expand the functionality of the exact cryptocurrency the developer wants to implement, in our case we've implemented both ProofOfConcept and BitcoinTransaction.

The transactionStorage actor is used to store, retrieve, and clear both entities and transactions. The functions are invoked in both the front-end and backendSorting canisters.

Lastly, the TransactionSorting actor is used to sort transactions based on the user selected field. Implementing a mergesort algorithm to sort the transaction, then inserting them in their respective arrays. In that manner, the sorting only happens if the original transaction storage array and the respective sorted array are not the same size. Otherwise, the already sorted array will be returned to the user saving computing costs.

Frontend Architectural design:

Using svelte for the front-end framework, several svelte native features are utilized to create an architectural sound approach to the requirements of this hackathon.

The vite.config files are configured so that they process environment variables such as the host or network based on where it's hosted either locally or on the internet computer.

The updates.types.mjs updates the type declarations placing them in their respective file path.

Note** Both the vite.config.ts and updates.types.mjs are from the original svelte starter repository from the dfinity examples repository.

The package.json contains scripts to generate, build, and deploy the canisters both locally and on the internet computer.

The layout.svelte contains the header and footer separated by the slot. Additionally the light/dark mode is initialized in the layout.svelte applying global changes to the state of the application. The web application contains one page with several components added to make the code easier to read and by componentizing some of the sections they can be imported in another page in the future.

The lib folder in svelte is used to save several components that are reusable. It is separated into four folders.

Components/UI contains the Shadcn components that can be imported to the required page. Additionally, it contains all the componentized pages such as the transaction explorer and the entity viewer.

The Motoko imports is a file that imports the Motoko backend canisters and are exported so that they can be used globally throughout the application.

The data folder contains svelte stores that are globally accessed variables. While not the best option architecturally, currently the entity data is also stored in the data folder. It contains a JSON array filled with the entity object such as their name, amount raised, image, and description.

Lastly, the images folder placed is placed in lib rather than static. This is done to import images dynamically for performance reasons. By placing the images in lib vite adds a hash to the image which caches them indefinitely until the key changes eg.. the picture is changed. This is done so that the user doesn’t request the same images each time.