

# Five West Assessment

Ndakondja Ndasindana Shilenga

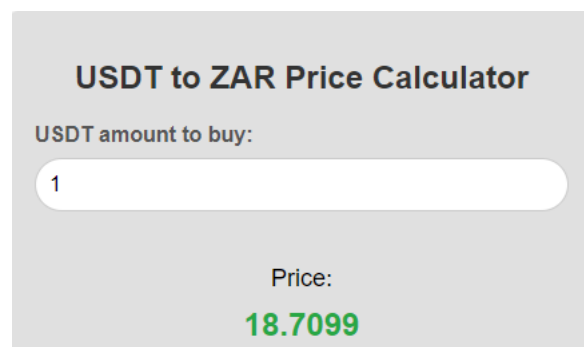
ndasindana@gmail.com

## Part 1 Implementation

The backend is implemented using FASTAPI as recommended and I connect to the VALR WebSocket API to receive USDT/ZAR price pair. The orderbook data fetched from VALR is stored in a dictionary with the two keys "bids" and "asks" each mapped to a SortedDict data structure from the SortedContainers library. The data structure was chosen for its efficiency in maintaining a sorted order to enable quick data retrieval and updating prices in the order book.

To calculate the price for a given USDT quantity, the `get_price_from_orderbook` function iterates through the ask prices, providing an accurate cost based on the current market state. The method to calculate price is exposed via a GET endpoint, which the frontend queries to display the latest price. A GET request is used here because it retrieves data without modifying the server state, making it suitable for fetching current price information. In addition to the price endpoint I also added two endpoints startup event which is configured to start the WebSocket connect to VALR when the application starts. I also have the root endpoint which I used to verify the server (backend) is running.

The frontend is built in react and I use Axios for HTTP requests to enable communication between the front and backend. React's state management and component reactivity allow for a seamless user experience, where the displayed price reflects the most current data from the backend. The choice of React and Axios ensures that the frontend remains dynamic and responsive, effectively displaying real-time price information based on user input.



USDT to ZAR Price Calculator

USDT amount to buy:

1

Price:

18.7099

Figure 1. USDT to ZAR price calculator

## Part 2 Implementation

Building on the existing backend, the new feature focuses on rebalancing a crypto index fund with an asset cap. To achieve this, I added a POST endpoint that accepts parameters such as the asset cap, total capital in ZAR, and a list of coins with their symbols and market caps. A POST request is appropriate here because it allows the client to send complex data structures to the server in a secure

and encapsulated manner. The backend fetches the USDT price for each coin from Binance using the provided API endpoint. These prices are converted to ZAR using the USDT/ZAR rate obtained in part 1. To ensure type safety and validation of the incoming data, custom models were created to store the data sent from the front end:

- Asset: the symbol (string) and mcap (float) for each asset.
- RebalanceRequest: asset\_cap (float), total\_capital (float), and a list of assets (List[Asset]).
- RebalanceResponse: the symbol (string), price (float) amount (float), usd\_value (float), and final\_percentage (float) for each asset.

These models help enforce the correct data structure and types, reducing errors and making the code more maintainable.

The frontend remains a React application that interacts with the updated backend. The frontend sends a list of assets and the market cap as well the asset cap and total capital to the backend using Axios for HTTP requests, ensuring efficient and asynchronous communication. The results, are then displayed to the user as can be seen in figure 2.

Price:

18.7099

Rebalance Fund

Symbol

Market Cap

BTC

20000

Remove

ETH

10000

Remove

LTC

5000

Remove

Add Asset

Asset Cap:

0.5

Total Capital (ZAR):

1000

Rebalance

Rebalance Results

Symbol	Price	Amount	USD Value	Percentage
BTC	56250.03	0.000475	26.71	50.00%
ETH	2986.40	0.005751	17.17	32.14%
LTC	64.71	0.147435	9.54	17.86%

Figure 2 Part 2 Rebalance fund

### Part 3

- a) Figure 3 is an image of my ER diagram (I auto created this in SQL Server Management Studio by creating my tables first.

The tables and their attributes as well as purpose are as listed below

Funds(Fund\_ID, Fund\_Name, CreatedDate, CreatedByID, ModifiedBy, LastDateModified)

- This table stores information about the various funds being managed, including their names and creation/modification details.

Assets(Asset\_ID, Symbol, CreatedByID, CreatedDate, ModifiedByID, LastDateModified)

- This table holds details of the individual assets that can be part of the funds, identified by their symbols.

Fund\_Assets(Fund\_Asset\_ID, *Fund\_ID*, *Asset\_ID*, MarketCap, CreatedDate, LastDateModified)

- This table represents the relationship between funds and their constituent assets, including the market capitalization of each asset within a fund.

PriceHistory(Price\_History\_ID, *Asset\_ID*, Price, PriceDate)

- This table stores historical price data for the assets, capturing how the prices have changed over time.

Rebalances(Rebalance\_ID, *Fund\_ID*, CapturedDate, TotalCapital, AssetCap)

- This table logs each rebalance event for the funds, including the total capital and asset cap used during the rebalance.

Rebalance\_Values(Rebalance\_Values\_ID, *Rebalance\_ID*, *Asset\_ID*, FinalPercentage, AmountAllocated, Value)

- This table records the final allocation details for each asset in a rebalance event, including the final percentage, amount allocated, and the value

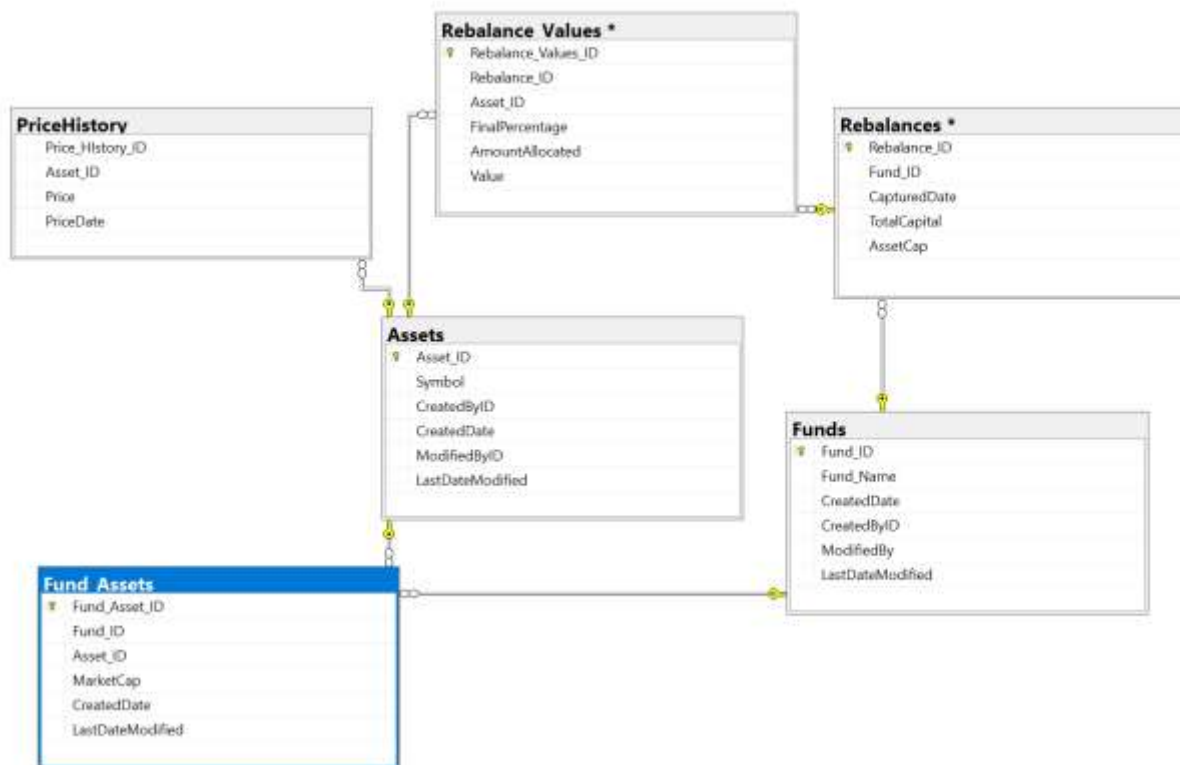


Figure 3 ER Diagram

b) Figure 4 is an illustration of my sequence diagram, the sequence flow is described below

#### User: Create Fund (if necessary)

Frontend: User initiates the creation of a fund by entering fund details and assets.

Frontend -> Backend API: Send a request with fund details and assets.

Backend API -> Database: Store fund details and assets in the database.

#### User: Request Rebalance

Frontend: User requests to rebalance the fund.

Frontend -> Backend API: Send a request to rebalance.

Backend API -> Database: Retrieve fund details from the database.

Backend API -> External API: Fetch current prices of assets from the external API (e.g., Binance, VALR).

External API -> Backend API: Send the current prices to the backend.

Backend API -> Database: Retrieve stored prices from the Price History table if necessary.

Backend API: Calculate the new target allocations based on market capitalization and asset cap.

Backend API -> Database: Retrieve current cap from the Fund Assets table.

Backend API: Compare target allocation with current holdings and determine rebalancing actions.

Backend API -> Database: Update allocations and store rebalancing details in the Rebalance and Rebalance Values tables.

## User: View Results

Frontend: User views the rebalance results.

Backend API -> Frontend: Send the rebalance results back to the frontend for display to the user.

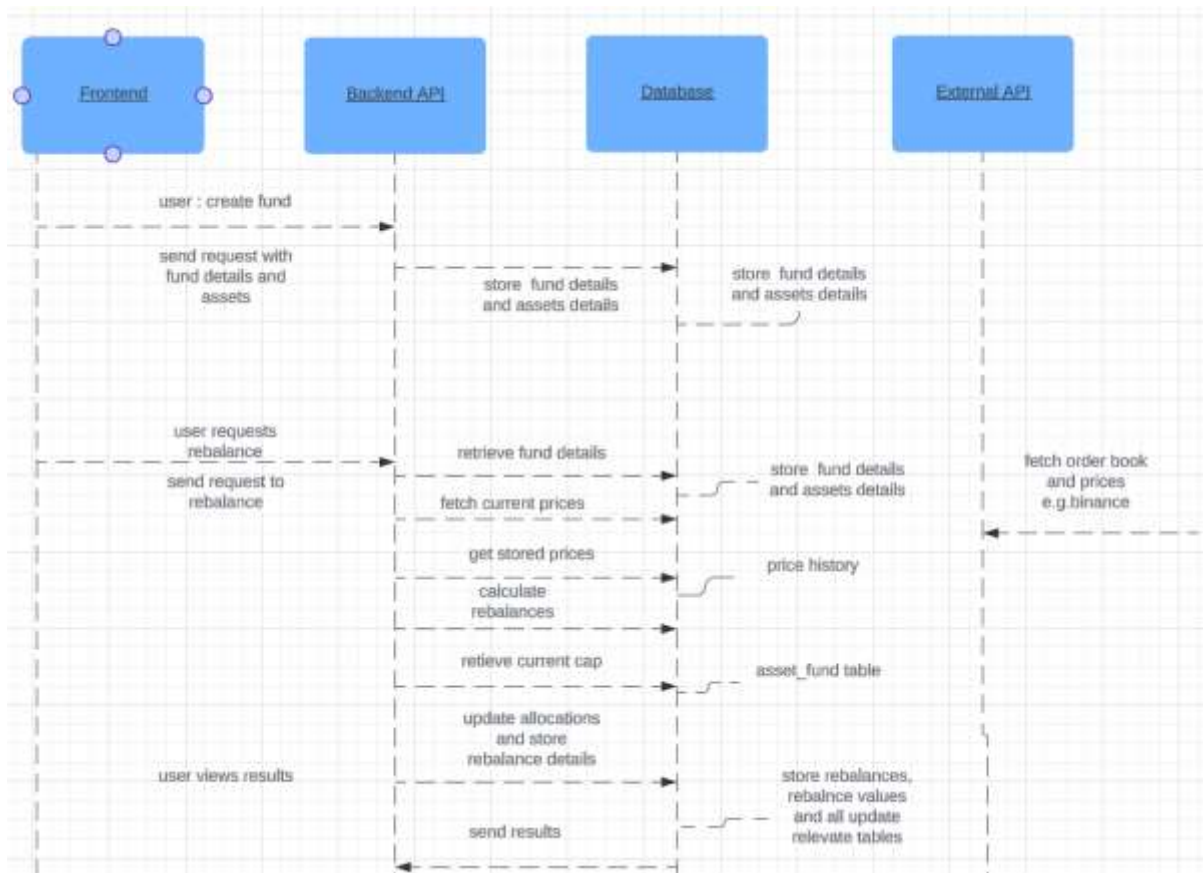


Figure 4 Sequence diagram (for a clearer view please click on [https://lucid.app/lucidchart/7831d601-ca9e-45cc-a881-71eade31491e/edit?invitationId=inv\\_b7b2977b-ab94-4624-aa39-089b6dfd4c63](https://lucid.app/lucidchart/7831d601-ca9e-45cc-a881-71eade31491e/edit?invitationId=inv_b7b2977b-ab94-4624-aa39-089b6dfd4c63))