Fetch Rewards Backend Exercise

Joel Patel

Nov 10th 2022

Tech used:

- Used **Go** with **Gin** framework to handle http requests-responses.
- Step to run it locally:
    - Please install Go.
    - Clone the repo.
    - Run "**go run .**"

Architecture Diagram: Please refer to the architecture.png file in the Documentation dir.

Components and their functions:

main.go:

- Func main is the entry point of the web service.
- Gets an env. variable for if provided, or uses "8080".
- Creates an API path router and enables logging.
- Sets up API paths and designates which functions will handle requests.
- Runs the web service at "http://localhost:{port}/".

data.go:

- Data.go of the db package will handle data in memory.
- It handles a slice (array/list from other languages) of Transaction type and a mapping from string payer to integer points.
- The Transaction type will have a string payer, an integer points and a timestamp in UTC.
- The **InsertTransaction(...)** function handles insertion into the Transactions slice.
- It stores data in order by timestamp format. If it were a SQL database like postgres, then one way to store it would be just insert data to the table with *clustered index* on timestamp field.
    - The function also checks if the value is negative, in which case it'll remove the points from the previous entry and keep the latest timestamp using the **SearchAndDestroy(...)** function. Furthermore if some points are remaining from old entries then the insert transaction will handle inserting into proper place.

- **RemoveTransactions(...)** removes transactions from a slice of indexes which were previously marked for deletion. You can consider this as batch removal.
- **GetTotalPoints(...)** sums up all the points from the Payer mapping. It is used to check whether the user has enough total points compared to what they have requested to spend.

addTransaction.go: (/add)

- It has a function, **AddTransaction()**, which returns a handler function as it's required for the Gin framework.
- The function takes the request body and converts it into a Transaction struct variable. If there's an error, it lets the requestor know about it.
- Validation is also performed, but it is very minimal in my implementation, basically just that the fields are required.
- Then checks if adding this transaction will cause the points for a payer go negative.
- If everything's good then updates the points of a payer and inserts data using the **InsertTransaction** function of the **db** package.
- And sends a 200 status code with a JSON object response to the requestor.

spendTransaction.go: (/spend)

- It has 2 struct representing the request body and another to track which payer spent how many points.
- The **SpendTransaction()** function checks whether the user has total points among all their payers or not.
- The function then loops over the transactions and starts spending points if it is possible to do so based on the transaction's points.
- Updates values and tracks which rows from Transactions need to be deleted.
- If everything's ok and the amount was spent and the function responds with a 200 and all the tracked spendings to the requestor.

getAllTransactions.go: (/alltransactions)

- Sends a response with all the transactions.
- Use for debug purposes.

getAllPayers.go (/all)

- Sends a response with info. about all the payers.

⇒ Thank you recruiters for giving me this opportunity. I hope I met your expectations in this test. I’m excited to hear back from you.