Project Update

Due 4/12/2020

The Planning service performs two separate calculations depending on which request it receives. The request comes via the JSON message format listed for the planning service. There two calculations are performed in their own respective functions. Those two functions (calcTimeFrame() and calcEndBalance()) have been written and very mildly tested. The output format for the functions to send a JSON message have not been implemented yet.

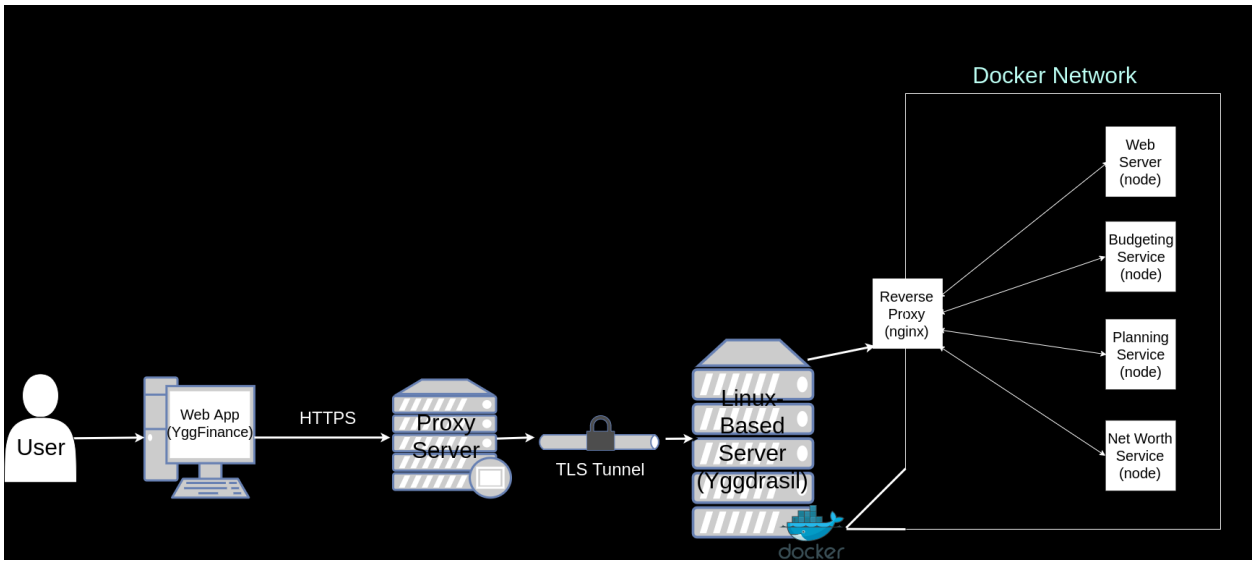
The Monthly Budget service has been partially implemented. The service still needs to be designed that receives a csv-process request. The function parseCSV() has been partially implemented. The function reads in and parses a csv file. The challenge is filtering out only the columns needed when no header is present. We still need to implement a way to filter down the original csv to only the columns that are needed to create transactions. Once that is done implementing a way to add the transactions into an array a pass it via JSON message format will be the final step.

The Net Worth service has been partially implemented. This service will receive the user's assets and liabilities in the form of an object with key value pairs. The calcNetWorth() function iterates through the assets and liabilities summing the values. The calNetWorth() function then subtracts the assets total from the liabilities total returning a number. The function has been implemented, but it still needs the JSON message format for the output to be implemented fully.

Docker will still be used to create containers and images for each of the three services. Docker will direct the incoming request and outgoing responses to the appropriate sources. Docker still needs to be configured specifically for this application. This will include creating a Dockerfile, DockerCompose file, and setting up a reverse proxy nginx container to serve as an abstraction layer and direct requests to different application endpoints to their respective services inside the Docker Network. Currently Test and practice Docker images and containers have been set up, but the current work on YggFinance has not been Containerized and Deployed into Docker Images yet.

The team member in charge of the Server setup and configuration has just recently finished setting up the Yggdrasil server. After much consideration, we decided that the learning curve and potential hidden costs of online hosting providers to be too high of a risk at the current stage of this project. Instead, we decided to direct our efforts towards setting up a dedicated home server (which the member in charge of this has ample experience in) and then looking for a solution for making this publicly available. Luckily, after much research, we discovered a set of solutions that fit all of our project constraints allowing us to securely tunnel a port on the server's localhost to a proxy service configured to securely forward internet traffic to

and from the server. The initial solution we discovered was a service called ngrok, however it quickly became apparent that the free tier of their service was too limited for what we would prefer. As a result, after more research, we discovered a Free (in both senses of the word) and Open Source Software (FOSS) solution that performed the same function while allowing us to request access to any third level domain name of our choosing on the server (given it isn't currently in use when requested). This solution is called Localtunnel and allows us to very quickly and easily connect our server to a publicly accessible domain. As of the writing of this document, the server is currently up and accessible at https://yggfinance.loca.lt/. As the YggFinance services have yet to be containerized into docker containers, the server is currently just running the standard docker getting-started tutorial image to demonstrate that it is properly working. The updated network topology diagram is as follows:



The actual Server hardware configuration consists of an Emachines ET1331G-07w running Ubuntu Server 20.04.2 with docker, docker-compose, npm(Node Package Manager), and ruby installed using apt(Advanced Package Tool), localtunnel installed via npm, and localtunnel-restarter installed via a ruby gem. For convenience in local administration, the x-org window manager, i3 desktop environment, and OpenSSH server were installed and configured as well. Due to our choice of technologies, no network or router port configuration was required for this server past a standard ethernet connection into a basic internet router (of course connected to a modem on a standard 50/5 home internet connection).

One main unresolved issue is how to handle the csv upload for the monthly budgeting service. The challenge is that our program is designed for the user to select which columns correspond with the appropriate elements of a transaction. So far, the CSV parsing libraries we have looked into do not have a great way to implement this when there is no header. We would like to implement this feature so that we can have broader usability with our application. However, for simplicity, we may have to implement the csv portion solely for a specific bank or credit card company's format.

Another big piece of implementation is testing the application. Once the front end and back end are able to communicate testing can commence. Each service is operating separately, so there will be a set of test data for each service.

Currently the front end UI is very close to being able to begin testing for the Monthly Budget and Savings Planner functions. As the group member working on the UI has gained a better understanding of React, MaterialUI, and their respective limitations, changes have been made to the assortment of interactive elements as needed. At the current level of progress, one such change has been that we are no longer planning on implementing an edit table for the user data inputs on the monthly budget page. The EditTableAPI has not been updated since 2017, and we are worried about possible conflicts with the more recently updated MaterialUI. The solution to this will be to simply use user inputted text fields. Though maybe not as visually pleasing, it will serve the same function.

The interactive elements on the page will change as we need them, namely the ability to reconcile on the monthly budget page. If we cannot figure out how to read in and compare data from a CSV file, then there will be no use for the reconciliation mode. The monthly budget planner page will still have its uses, however.

The most difficult problem left to tackle in the UI is data transfer and management. Making sure the front end can identify data properly and communicate with the back end may prove to be challenging at first. We also hope that the net worth page data handling can be done through the front end alone. We will not know if this will be an issue until testing for the page takes place, though luckily the function only does simple math and has no need to save or load values upon page reload.

After testing with data, if the app functions properly with extra time we would like to make visual changes to the UI to clean up the user space. Though this is not necessary for the app to function, making it look professional is a goal if given the opportunity. This may be achievable as we are on pace to begin testing with data this week. Plans to “professionalize” the app would include things like app logos, editing the positioning of items, and changing theme styles to be more customized.