# Project: YggFinance

Group members: Blake Hudson, Alan Holman, and Austin Kerr

## Table of Contents

# 1 Introduction

YggFinance is a personal finance tool for anyone looking to improve their personal financial health or accomplish long term goals. YggFinance helps users set goals, track progress, and provides tools to stay on track. It also assists users in setting goals by providing a calculator to look at future projections of their investments or starting with a target goal and figuring out how much they need to save each month. Ygg finance also helps users track some of the key performance indicators of personal financial health like net worth, monthly spending and savings goals.

One metric that is great for getting a broad overview of one's finances is net worth. YggFinance actually guides the user in calculating their net worth, teaching them how it's calculated at the same time. This adds value to the user by not only helping them track their net worth, but teaching them at the same time. Another big question that folks have when it comes to finances is "when can I retire?". YggFinance allows them to put information about their current investments and get an idea of what their investments will be worth in the future. This helps them figure out how much they need and when can they meet certain goals, not just retirement. The biggest value YggFinance adds is the monthly budgeting feature. Psychology plays a bigger role in an individual's success with their finances than math does. YggFinance

aims to take away as much friction as possible from tracking expenses and budgeting, making it more likely a user stays with their plan.

# 2 Technology

During the making of the YggFinance Web Application we utilized a large variety of technologies.

## 2.1 Architecture

The first thing we had to decide on before we got to putting together a design for our project was what kind of architecture we were going to use. At its base, we needed to decide between a Monolithic and a Distributed Architectural Pattern. Because we wanted to develop a Web App, this necessitated some kind of Distributed Architecture as completely Monolithic Architecture doesn't support this kind of project. Rather than settle for a standard Client/Server Architecture, we wanted to instead gain experience utilizing an architecture that promotes better scalability and deployability than the Client/Server Architecture which is known for causing frequent issues in these areas. As a result we decided to go with the **Microservices Architectural Pattern** as it alleviates many of these issues while showing amazing results for many top companies who have migrated their systems to this architecture.

Because we have YggFinance with the Microservices Architecture Pattern, we had to decide on which technologies to use for implementing the various components of this Architecture. While there are many options available to meet the high scaling needs of enterprise development, most of these advanced options such as Kubernetes and AWS ECS/EC2/CLI were all fairly far outside of our expertise and proved to be mostly overkill for the scale of our project. In order to handle the isolation of Services within the Microservices Pattern, we opted to use **Docker Containerization**, as it is fairly easy to pick up, provides us with flexibility in server setup and application deployment, and was one of the most universally used technologies we had researched. For Container Discovery and Orchestration, rather than using a more advanced alternative, we decided to stick to Docker's bundled solution called **Docker Compose**. Docker Compose allows us to easily configure the settings of and connections between various docker containers in a single easily readable YAML file.

## 2.2 Version Management

In addition to these technologies, we also decided to utilize **DockerHub** as our **Container Image Repository** in order to host the built and released container images for each of our services. This makes it possible for anyone to fully deploy our application with only the docker-compose.yml file located in the release folder of the repo as opposed to having to build the project from the source code at deploy time. Using Docker Containerization for this is what

allows us to be flexible with our choice of server hosting platforms without being tied to any one platform and is the primary reason for our choice in using this technology.

For **source code versioning**, we decided to use **git** with some team members utilizing **TortoiseGit** for a windows shell interface with the repository hosted publicly on **GitHub**. The primary reason for using git repositories and hosting on GitHub is due to these technologies being part of the project constraints provided to us. This said, we would likely have chosen to utilize the same technologies had it not been a project constraint due to the widespread use and collaboration advantages that these software provide over potential alternatives.

## 2.3 Programming Languages

Getting down to the application itself, the primary programming language we decided to use for the application code was **JavaScript**. While we had originally considered having each developer code their own services in the language they were most familiar with to lessen the learning curve, we quickly discovered that while our architecture makes integrating multiple languages incredibly easy, the learning curve of the HTTP request libraries for certain languages proved too high for the short span of time we had to complete this project. As a result, we opted to stick with only JavaScript for both the backend and frontend due to its flexibility, simplicity, and fairly low learning curve. For building and running the JavaScript applications, we are utilizing **Node.js with NPM(Node Package Manager)**. We chose Node primarily because, aside from maybe some obscure side projects, Node is the defacto standard for executing JavaScript code outside of embedded HTML in a Web Browser. When it comes to NPM, we could have used a different dependency management solution such as yarn, but for simplicity's sake we decided to stick to the default bundled solution.

## 2.4 NPM Dependencies

Within our application, there were several libraries that we made use of that were added to the dependencies list in each service's package.json file. Among the API Services (Budgeting, Planning, Net Worth), the one shared dependency that we have is the HTTP Server Library **express**. While JavaScript has a built in HTTP library (aptly named http), it tends to be fairly low level and somewhat un-intuitive to use. As a result, we decided to utilize Express instead because it greatly simplifies and abstracts away much of the complexity that the base http library comes with. We also chose Express as opposed to other alternatives due to its popularity in industry use.

In addition to Express, the Budgeting service also includes two additional dependencies to support its functionality. The first additional dependency is an additional middleware for express for parsing multipart form data request bodies called **multer**. The second one is a basic csv file parsing middleware aptly named **csv-parser**. We chose multer primarily because it is express's official middleware for multipart form data parsing, though csv-parser was a bit different. While there are a plethora of different libraries for parsing csv files in JavaScript,

overall it was the lightweight simplicity of csv-parser that caused us to choose it over the more bloated alternatives such as papaparse.

For the Frontend Web Application, we opted to utilize the React library (**react** & **react-dom**) with **material-ui** components for consistent styling across the entire Web Application. Interestingly enough, the main reason we chose React over alternative frontend web frameworks was because we liked the look and feel of components provided by Material-UI, which is a framework for React. While it is true that we could have mixed and matched frameworks, we decided to keep things simple and just stick to React with Material UI for styling. Keeping with the theme, while Material UI offers many beautiful elements to work with, putting together an easily editable data entry table proved to be a bit of a struggle due to our inexperience. As a result, we discovered an extension of the Material UI elements providing just the component that we needed as the **material-ui-table-edit** library. Because we were not sure where to start or how to set up a React-based web app project, we opted to use React's officially supported create-react-app tool to bootstrap the web app part of the project. As a result, various modules from the **react-scripts**, **testing-library**, and **web-vitals** libraries were added as dependencies. We do not directly use any of these out of inexperience and lack of project time to learn, however, they are referenced in various areas of the web application's backend code as part of the project setup process of the create-react-app tool.

## 2.5 Base Docker Images

Initially, before we had properly containerized the project with docker, each of our team members were required to install NPM in order to install dependencies, build, and run each service in the application. However, now that we have properly containerized each service with docker, the developer no longer needs to have npm installed on their local system. Instead, all the building and executing of the services is performed within the service's container image itself, eliminating all dependencies on the base machine's hardware configuration aside from docker itself. In order to do this, we utilize a Base Container Image for building the final service's container image that contains the individual services. Across all of the Services in our project, we utilize **2 Base Container Images** to build all of the container images for our application. We use the **Node:14-alpine** Base Container Image for the API Services & Tunneling Utility and the **Nginx:alpine** Base Container Image for the Reverse Proxy & Web App (backend). The Node:14-alpine base container image runs on alpine-linux and is installed with npm and node for building and executing our backend services. The Nginx:alpine base container image also runs on alpine-linux though is only installed with Nginx, which is a lightweight easily configurable web-server. We chose to use Nginx over other alternative web-servers such as apache due how commonly it is used for javascript based web deployment as well as how often it is used in conjunction with docker. We chose to specifically use the alpine-linux variant of the docker images due to the space saving advantages that alpine-linux based images have as opposed to the basic debian-linux based ones (about a 10x decrease in image size [~1 GB -> ~100 MB]).

## 2.6 Scripting & Configuration

By utilizing docker to remove the need to configure the system deploying the software, we are really just instead moving the configuration of the server logic into the containers themselves. This means that we have to utilize various configuration tools and scripting in order to set-up the server infrastructure within the docker containers themselves. In the case of Docker, we utilize a **Dockerfile** to configure how to build the docker image for each service, and, because we are using Docker Compose as our container orchestration solution, we use the **docker-compose.yml** in order to configure the connections and execution arguments of each of the containers. Because we have chosen to use Nginx as our web server, we configured the **nginx.conf** to set up the reverse-proxy (the web app (backend) is properly served with only the default config so no configuration necessary). Now, while Docker Compose allows us to configure most of the build and deploy process in the docker-compose.yml file, some configuration tasks are easier handled using a simple script file. While we could use a variety of scripting languages in order to accomplish this, we opted to utilize **Bash shell scripting** for this task as it was the most straightforward way to perform the functionality needed without requiring additional dependencies on most server systems.

## 2.7 Application Persistence

While most systems will opt to use a server-side data storage solution such as event stores, data stores, or relational databases due to the advantages these provide in accessibility and auditability, we instead opted to utilise the **localStorage** client side storage solution. While Server-Side storage methods allow for many benefits such as more flexibility when working with client data, these benefits come at the trade-off of certain characteristics such as Security, Privacy, and Legal concerns. When client data is stored on a server, in order to ensure proper authorization and authentication, additional systems to intentionally differentiate users requests from one another must be implemented thus negatively impacting user Privacy. If these systems are not properly implemented then it has the potential to compromise not just one client's data, but instead it compromises the data of all the clients utilizing the server spelling Privacy and Security Concerns. When instead storing relevant client information on the client side and only the relevant server information on the server side, it eliminates the need for differentiating generic user requests and ensures proper authentication (as there is no client data stored on the server to authenticate for) and authorization (as generic user requests are grouped into one authentication group) increasing overall Security and Privacy.

## 2.8 Communication Standards

We decided to keep the communication standard that we use between the services in our application as simple as possible by sticking to **HTTP** requests with **JSON (application/json)** as the message encoding. This said, because one of the features our application supports requires sending a csv file to the budgeting service, we had to adjust the message encoding type we utilized for sending that one request to be of encoding type **Form Data (multipart/form-data)** instead (the response remains as JSON however).

## 2.9 Proxy Tunneling

While we could have aimed for deploying our application to a single hosting platform, after many issues with pricing and learning curves of various hosting platforms such as AWS or Digital Ocean, we decided to direct our efforts instead towards self-hosting. Self-hosting itself has many issues that we ran headfirst into such as the security risk of opening router ports, the cost of acquiring a domain address, and ISP restrictions on obtaining Static or Public-facing IP Addresses. Overall, these are not very easily solved issues if you approach them headfirst. Instead, we opted to sidestep these issues entirely by utilizing an Open Source reverse proxy CLI utility called **Localtunnel**. This utility allows you to tunnel any single port on your localhost (configurable to be a port on any IP technically) to any subdomain of your choosing (that isn't in current use) on a community maintained proxy server (loca.lt, localtunnel.me, etc. [the server is also open source and self-hostable/configurable]). This allows us to provide public access to our application without needing to reveal or care about the server's IP or open any router ports. This kind of tool is often used for API testing in development environments and as a result there are many more mature and stable alternatives such as ngrok and pagekite. Unfortunately, while these alternatives are almost certainly functionally superior in every way to localtunnel, they both are incredibly limited for free users and don't offer a free choice of subdomain names unless you are a paying user which is why we opted to use localtunnel instead.

## 2.10 Development Tools

While the application itself is largely independent from most development environment conditions, it is still important that we discuss the tools and environments that we utilized in the making of this application. To begin, development and testing took place on a mix of operating systems including **Windows 10**, **Manjaro** Linux, and **Ubuntu** Linux. There was no particular reason for the variation in operating systems other than the convenience of each of the developers.

On the other hand, despite each developer starting on different IDEs, we all eventually converged into using **Visual Studio Code** as our primary IDE of choice after learning more about the technologies being used. The primary plugins of note in VSCode that we took advantage of were **code-spell-checker** for spell checking**, markdown-all-in-one & vscode-markdownlint** for markdown management**, vscode-versionlens** for dependency tracking**,** and **vscode-docker** for docker integration. The primary reason we all moved to using VSCode for our IDE has to do with various difficulties we encountered with other IDEs such as Netbeans, Eclipse, and Visual Studio messing with project directory structures and overall not providing a very clean or useful interface for managing our project. For more mature projects, we would likely consider configuring another more full featured IDE for our needs, but the simplicity of VSCode allowed us to work on the project without having to worry about IDE Project conflicts.

## 2.11 Documentation Tools

For documenting the project in the GitHub Repo, we are using **Markdown** as it is the defacto standard documentation format for Repo documentation. When it comes to the Various Deliverable documents and presentations we put together over the course of this project we have exclusively used **Google Docs and Google Slides** due to the ease of collaboration of these tools over standalone document editors. While there was the option of utilizing the Microsoft Office Online suite, ultimately, our team was more familiar with using Google's offerings so we stuck with what we were most familiar with. When creating the various diagrams present in our documentation, we exclusively used **app.diagrams.net** due to incredible flexibility of the tool while being free to use open source software. Most other diagramming tools did not seem to provide the same range of functionality that app.diagrams.net could, with even some licensed tools such as Microsoft Visio falling behind despite being a paid software.

## 2.12 Team Communication

For team communication, we almost exclusively utilized a private **Discord** server only using **Zoom** for meetings when in attendance to the University Course. The primary reason we opted to use Discord as opposed to Email or SMS Messaging was because of our familiarity with the technology, and the cross-platform, easy-to-use, reliable nature of the software. From our experiences using both software for team communication and meetings, we encountered many situations where Zoom was under-performant and buggy compared to Discord in the same setting causing us to have to move either part or all of our meetings over to the Discord Server. The primary advantage that discord provides over various other communication tools we have looked into is that it has allowed for easily communicating, leaving messages, giving progress updates, posting resource links, helping debug issues with a screen share or voice call, and having light conversation with other team members all in the same communication tool.

# 3 Design

The YggFiance Web Application is designed with the frontend as a Reactive SPA(Single Page Application) and the Backend designed as a distributed collection of containerized microservices. To get a better idea of what this means for the design of our system as a whole, please reference the System Topography Diagram in the appendices (A.2).

## 3.1 User Interface Design

The overall design consists of four pages accessible via tabs in a  top menu of web app's dashboard. The top menu will be default material UI TabPanel available to the user at all times (A.4.1). Clicking on the appropriately labeled which represents a service that YggFInance provides. The four tabs are the welcome page, savings planner, networth, and monthly budget.

The welcome page explains the purpose of YggFinance and what each page does. The savings planner page gives access to the two different planning modes that the user can interact with by inputting data into the required fields. The networth service guides the user through entering the appropriate data to calculate their net worth. Finally, monthly budget allows users to create budget categories, input transactions, and upload a csv file to compare transactions from their bank statement to ones they have manually entered.

### 3.1.1 Welcome Page

The welcome page simply consists of a large amount of text explaining what each page of the application does. Implementation wise, it is simply text directly in a Tab Panel of the NavBar for the application. The Tab Panel works off of three base functions. These functions are TabPanel(), a11yProps(), and LinkTab(). These serve to set up the functionality of the TabPanel item, while the NavTabs() function is the main function on the file to implement our specific navigation tab. A screenshot of the welcome page with the NavBar on the top of the page can be viewed in the appendices(A.4.1).

### 3.1.2 Saving Planner Page

The Savings Planner tab will give you access to our Savings Planning function. This function has two selectable planning modes: Contributions and Savings Goal. While in Contributions mode(A.4.2.1), the planner will ask you for an input of an initial investment, average rate of return on the investment, expected final savings goal, and how many years you expect to let your investment grow. Below these fillable text boxes will be a button to calculate results. To the right of your entries, you will be able to view the calculated results and data including your savings ending balance, the time frame of saving, the amount you started with, the amount of contributions needed monthly for the investment, and total interest made on the investment. If the selectable slider button is engaged, the Savings Planner will change to Savings Goal mode(A.4.2.2). Inputs and outputs are similar, but this page solves for an amount of savings you wish to reach instead of how much you need to contribute per month. The time frame output on this mode will let you know how long it will take to reach your specified savings goal with your specific conditions that were applied.

Local storage data is separated by inputs and outputs. The inputs include six fields associated with a number value which has a default value of zero: savingsGoal, initialInvestment, timeFrame, avgRateOfReturn, monthlyContributions. The inputs also include a field with a number value with a default of one called “planningMode”. The outputs include five fields associated with a number value which have a default value of zero: endBalance, timeFrame, startingAmount, totalContributions, totalInterest. These values will remain in their default state until the user enters inputs. Page data is updated as a copy of the local storage data. The number values of savingsGoal, initialInvestment, monthlyContributions, endBalance, startingAmount, totalContributions, and totalInterest are all representative of US dollar amounts. The number value for timeFrame is representative of years. The number value for avgRateOfReturn is representative of a percentage formatted in decimal form. The number

value for planningMode is representative of the identifier value used to show which savings planning mode the page is currently in.

All the elements of the UI for the page are in a single, large grid element with three smaller grid elements inside of it. In the leftmost grid element are four textfields. In either planning mode, three of the textfields are labeled “Initial Investment (USD)”, “Average Rate of Return (%)”, and “Years to Grow”. The fourth textfield is labeled “Savings Goal” while in contributions mode and “Monthly Contributions (USD)” while in savings goal mode. In the rightmost grid element is a table including all of the output data. These values are labeled “End Balance”, “Time Frame”, “Starting Amount”, “Total Contributions”, and “Total Interest”. The lowermost grid element consists of a header for the mode selection switch labeled “Planning Mode” and the switch itself. The switch has a left label of “Project Contributions” for contributions mode and a right label of “Project End Balance” for savings goal mode. The switch being in the leftmost position gives the planningMode item a value of zero, while it being in the rightmost position gives it a value of one. This value is used to tell the page and the back end service which mode the service planner page is being requested.

### 3.1.3 Monthly Budgeting Page

The Monthly Budget tab will give you access to a budgeting tool. This tool will allow you to make a budget for each month by using an interactive table. Buttons will be available to save changes, start a new month, and to select which month you would like to view from previous entries. The collapsible table will have editable regions for categories you plan to budget, the amount you have spent to date, and the total amount you have budgeted to spend for the month selected. Below this table is a field to enter transactions. These transactions require a merchant name, the amount spent, and the date of the transaction. You may save changes after entering a transaction, which will process the transaction into your month's data. An example of the UI for the service can be found under “Monthly Budget Page UI” in the appendices(A.4.3). After finishing the transactions for your month, or for past months, you may choose to reconcile transactions by selecting the reconcile button. This will convert the page to reconciliation mode(A.4.3.2), which will give you the option of uploading a CSV document of transactions from your bank statements. The page will allow you to view self-recorded statements and compare them to statements from the bank. This will allow you to view any discrepancies and possibly find fraudulent charges. Totals for amounts spent in both the self-reported and bank-reported sections will appear at the bottom of the tables. You will be able to edit values in the tables in case there is some error in previously inputted data or in the bank statement. Once your totals match, your selected month is reconciled and saved into that month's history for viewing.

### 3.1.4 Net Worth Page

The Net Worth page will allow you to track your personal simple net worth. The page has editable text fields with labeled categories for assets and liabilities. There are prelabeled categories that may be helpful information to see how much of your assets and liabilities consist of those common categories. There are also categories labeled "other" in both sections to allow

users to enter in assets and liabilities that may be less common. When all values have been put in, the calculate button can be pressed to view  net worth in the box labeled "Net Worth". A figure of the Net Worth Page UI design is located in the Appendices under “Screenshot of Net Worth Page UI” (A.4.4).

Local storage data for the service is separated into three items. Assets and liabilities are both saved as an object, and net worth is saved as a number value. The assets object consists of six items with number values attached to them: realEstateValue, checkingAccountsBalance, savingsAccountsBalance, automobilesValue, other. The liabilities object also consists of six items with their respective number values attached: remainingMortgageBalance, consumerDebt, personalLoans, autoLoans, studentLoans, other. All of these numbers from the inputs are representative of US dollar amounts and are validated to be non-negative numbers only with a default value of zero. Page data is saved as an object labeled “netWorthData”, which is a complete copy of the local storage data.

Separated just like the local storage data, elements in the UI for the page are separated into three main groups. On the left side of the page is a vertically aligned grid element with six labeled textfield elements: Real Estate Value, Checking Account(s) Balance, Savings Account(s) Balance, Retirement Account(s) Balance, Automobile(s) Value, Other Asset(s) Value. These textfields display a value of zero until updated by the user. On the right side of the page is a vertically aligned grid element with another six labeled textfield elements: Auto Loans, Consumer Debt, Personal Loans, Remaining Mortgage Balance Student Loans, Other Liabilities. Below the two grid elements is a button element labeled “Calculate Net Worth” and a textfield labeled “Net Worth”. The button element is bound to a handle that, when clicked, the data entered into the assets and liabilities textfields will be sent to the back end, with the result from the back end being sent to the textfield labeled “Net Worth”. The Net Worth textfield is only for output, and it is not able to be interacted with by the user.

### 3.1.5 Database Tables (LocalStorage)

Please Reference the Local Storage Layout in the Storage Appendix (A.5)

### 3.1.6 Required Data Files

The YggFinance Web Application does not depend on any third-party data sources for proper functionality. However, it should be noted that in order to take advantage of the bank-statement upload portion of our reconciliation feature, the user is expected to provide a csv file of their bank transactions. Regardless of the availability of this data source however, the reconciliation feature remains fully functional through manual entry without the need to send this data source to our server for processing.

## 3.2 Service Design

Each component and service in our system is containerized in it's own isolated container which contains all code and dependencies required to perform its single base function. Each Container is only accessible and discoverable by other containers in the system when the two exist on the same Docker Network as noted in the System Topography Diagram (A.2). Because Containers containing services each exist in isolation to one another, This means that any interruptions or malfunctions in any single service in the system will not propagate to the other services in the system. While this prevents bugs in containers used for services from interrupting the execution of other services in the system, failure in any container containing architectural support functionality has the possibility of bringing down access to multiple parts of the system (such as is apparent if the tunneling utility, or reverse proxy were to fail). For a better understanding of the classification of each of the containers running in our system, please reference the Module Table in the Appendices (A.1).

### 3.2.1 Budgeting Service

The budgeting service is designed to support the backend functionality of the Monthly Budget Page on the frontend. As is shown in the Budgeting Service UML Diagram in the Appendices (A.3.4), The budgeting service is designed to respond to HTTP Post requests that are made to the '/budgeting-service' endpoint on the server. It expects to receive a Request of the "multipart/form-data" type with the contents as shown in the UML diagram. Once it has received the request, it calls the parseRequest function passing down the Request and expects to receive a JavaScript Promise back The parseRequest function then parses the contents of the request and then calls the parseCSVtoJSON function with the parsed contents of the request body. The parseCSVto JSON function then returns a Promise that reads the contents of the uploaded csv file from the filePath, converts the data to a JSON format, adds it to a transactions array, and then calls the resolve function of the Promise with the complete transactions array. When the Promise resolves and is returned back up the calls stack to the start, the transactions array is retrieved and then wrapped in a JSON object that is sent back as the Response.

### 3.2.2 Planning Service

The planning service is designed to support the backend functionality of the Savings Planner Page on the frontend. As is shown in the Planning Service UML Diagram in the Appendices (A.3.5), The planning service is designed to respond to HTTP Post requests that are made to the '/planning-service' endpoint on the server. It expects to receive a Request of the "application/json" type with the contents as shown in the UML diagram. Once it has received the Request, it initializes a JSON Object for the Response, then determines what to do next based on the planningMode field in the Request. If the planning mode is 0, it executes the calcMonthlyContributions function. If the planning mode is 1, it runs the calcEndBalance. Otherwise, it logs an error with what the planningMode it received actually was. Regardless of which function executes, once everything else is done, it sends the JSON Object it created earlier as the Response.

The calcMonthlyContributions function reads the initialInvestment, avgRateOfReturn, timFrame, and savingsGoal fields of the request body, then calculates the endBalance, timeFrame, monthlyContributions, startingAmount, totalContributions, totalInterest, then writes all values to the Response JSON Object that was passed in.

The calcEndBalance function reads the initialInvestment, avgRateOfReturn, monthlyContributions, and timeFrame fields of the request body, then calculates the endBalance, startingAmount, totalContributions, totalInterest, then writes all values to the Response JSON Object that was passed in.

### 3.2.3 Net Worth Service

The net worth service is designed to support the backend functionality of the Net Worth Page on the frontend. As is shown in the Net Worth Service UML Diagram in the Appendices (A.3.6), The net worth service is designed to respond to HTTP Post requests that are made to the '/networth-service' endpoint on the server. It expects to receive a Request of the "application/json" type with the contents as shown in the UML diagram. Once it has received the request, it executes the calcNetWorth function, wraps the output number in a JSON Object, and then sends the JSON Object as the Response. The calcNetWorth function simply reads the assets and liabilities fields from the requestBody, sums up the fields in each one into totalAssets and totalLiabilities, then returns the totalAssets minus the totalLiabilities.

### 3.2.4 Web App (backend)

While the Web Application itself runs as a client side web application, in order for this to be properly served to the client when they access the web pages, it must be served from somewhere by something on our server. In our case, the Web Application is served to the client by a default Nginx:alpine web server container after being built by a Node:14-alpine container that is discarded once the build process is completed. Due to our Web Application's frontend being bootstrapped by create-react-app, the production building process of the source code compiles the application into a standard index.html that, on the default configuration, is automatically served by Nginx when any HTTP GET requests are received on port 80 of the container. In the scope of our application however, any GET requests made to the '/' endpoint on the server are redirected to this container.

### 3.2.5 Reverse Proxy

In order to ensure that each service in the application remains isolated from one another while being accessible at a single location, we opted to utilize a Reverse Proxy. The Job of the reverse proxy (also known as a load balancer) is to serve as an API gateway by detecting requests made to it and forwarding and redirecting them to the container containing the service that provides the api call requested. For example, requests made to application endpoints like "/budgeting-service" arrive at the reverse proxy, are redirected to the "/" endpoint and forwarded

to the correct port of the budgeting service's container. All of this is performed by configuring the nginx.conf configuration file that is used to configure an Nginx:alpine web server image when building the final image for deployment. To see the endpoints that the reverse proxy redirects and forwards, see the endpoints column of the Module Table in the Appendices (A.1).

### 3.2.6 Tunneling Utility

While the Reverse Proxy may be the single point of entry into the application itself, without additional server configuration, the container that the Reverse Proxy is running on is not accessible to the outside world. While we could simply map a port on the container to a port on the local server, this doesn't solve the issue of needing to expose the port on the local machine and ultimately opening a port on the network's router in order to allow external internet traffic through. There are a plethora of other things that need to be taken into account when opening a web server up to the internet that delves deeply into the field of network engineering. As a result of this complexity, while we could have utilized a cloud hosting provider such as AWS, the inherent hosting costs and high learning curve steered us into a different direction with a different solution. Rather than attempt to handle the complexities of configuring networks for public access, we opted to utilise a utility called localtunnel that allows us to simply tunnel a port on a running container to a community maintained proxy server.

The Tunneling Utility Container simply consists of a Node:14-alpine base image that is used to install the localtunnel utility through npm. In order to start localtunnel (and restart when it crashes from network instability), we are using a small bash script that we override the container's default startup command with. The bash script runs the localtunnel utility with a "--subdomain" option set to the $LT_SUBDOMAIN environment variable, the "--local-host" option set to the reverse-proxy container (referenced by just "reverse-proxy" as they are on the same docker network) and the "--port" option set to port 80 (the port that is exposed on the reverse-proxy container). The script then simply waits until the localtunnel command crashes and then restarts it automatically. All in all, this allows us to make our application accessible on the internet over a secure https connection without needing to open any router ports or even expose any ports on the local machine that is running the application (as the only exposed ports are to the containers themselves).

### 3.2.7 Database Tables (None)

YggFinance does not utilize server-side storage. All application persistence is performed locally on the user's machine.

### 3.2.8 Required Data Files

Just as with the client side web application, the backend does not depend on any third-party data sources for proper functionality.

# 4 Deploying

When deploying our application to a server, **only docker and docker-compose** are required to be installed and running on the server for our application to be executed properly. In order to deploy and run our application, the only file necessary is the **“docker-compose.yml” file located in the “release” folder** of our project (no other project files are necessary). Simply execute the “docker-compose up” command in the directory on the server containing this file and Docker Compose will pull down the pre-built, most recently released container images for our project from DockerHub, Configure the components, and then start the application. At this point, you should be able to access the application at “https://yggfinance.loca.lt” on any modern device that is connected to the internet through any modern web browser.

To be more precise about the url you can access the application from, you will be able to access the application from the “https://<LT_SUBDOMAIN>.loca.lt” address where <LT_SUBDOMAIN> is the subdomain name noted under the environment variables under the tunneling utility entry in the “docker-compose.yml” file (by default, this is set to “ygg-dev” for the docker-compose.yml in the src directory and “yggfinance” for the docker-compose.yml in the release directory). If the subdomain name configured under <LT_SUBDOMAIN> is being actively used by another party utilizing the localtunnel proxy server, then you should be able to see the random subdomain-url that you have been assigned under the logs for the tunneling-utility that appear when you run the docker-compose up command (or alternatively, you can simply change the config entry to a different subdomain name that isn’t actively being used).

# 5 Building

Because our application takes advantage of Docker Containerization, building each individual service in our application can be built with nothing other than **docker** installed on the build system. This can be performed in its simplest form by running the “docker build .” command inside each individual service directory in the src project directory. While this will successfully build the docker image from our source code utilizing the configuration in the “Dockerfile” located in the project directory, this single command doesn’t tag the images properly, nor does it set up the networks, or configure the services for deploying the application. For convenience in coordinating the build and configuration process of our application, we are utilizing the **docker-compose** utility, which usually comes bundled with docker installs. Instead of having to individually build and configure each and every service in our application, you can simply run the “docker-compose build” command from within the src directory and all services used in the application will be properly built and configured using the configuration in the “docker-compose.yml” file in the directory it is run from. From here, you can then run the application using the “docker-compose up” command and access it at “https://ygg-dev.loca.lt” (or whatever LT_SUBDOMAIN is as mentioned in the deploying section). For quick rebuilding as is

useful while debugging, you can also condense the building and running into one command by running the “docker-compose up --build” command in lieu of the other two.

For releasing a new build of our application for deployment to a production server, please take note of the make-release.sh bash script. While this script is configured for building and pushing the images to the tagged DockerHub Container Image Repositories (CIR) owned by shuruni (Alan Holman), if you wish to instead push the images to the DockerHub CIRs yourself, you will need to update the configuration in the docker-compose.yml to tag the images for your account, and then modify the login name in the make-release.sh script for authenticating the push. Take note that when running the docker-compose.yml in the release folder, you will need to update the tags of the images that are pulled down from DockerHub to reflect the account name you have used for pushing the built images to.

As a quick side-note to add additional clarity, our application does not utilize any databases and thus there is no need to create any for our application to run properly. If you wish to extend our application to make use of your existing database infrastructure, or to add additional databases to the application, due to docker containerization being used, this should have no effect on this stated build process aside from potential adjustments made to the build configuration files.

# 6 Known Bugs

There are a few bugs to mention on the first version of YggFinance. The first two bugs include our monthly budget page. Two of the main features included in the original requirements were the ability for the user to upload a csv file from their banking institution and select the columns containing the data that make up a transaction. The upload feature does not work at this moment.

The second major bug to mention also included the reconcile feature. Once the customer uploads their bank spreadsheet YggFinance would allow them to match all transactions from their bank with the manually entered ones, allowing them to account for everything. This feature has not been implemented since the csv upload has not been implemented either.

Another known bug across the entire app is the lack of data validation upon user entry. The user has the ability to enter bad data for some of the fields causing some features to not work properly for them.

Overall the app is also a bit laggy upon entering information. There is not a known solution at the moment, but it can make the app more difficult to use.

# 7 Future Work

Version 2 of YggFinance will add features to each for helping users set, track, and meet goals. The next feature will include added features to savings planner, a portfolio analyzer, and a way to link accounts for automatic tracking.

The added features for savings planner will allow users to calculate how much of a return on their investments are required to reach a certain goal. This feature helps users take the planning stage a step farther by starting with an end goal in mind and working backwards to figure out how to meet that goal by choosing an investment based on return rate. The portfolio analyzer will allow the user to enter in investment information like the balances of their accounts and how much they have in each holding. YggFinance's analyzer will then give the user a risk level of their overall investments based on the category and amount of each one. Finally the way YggFinance will further reduce friction for the user, in following their developed plan, is through automatic tracking. YggFinance will allow the user to safely link their accounts from their credit card and banks to pull transactions automatically. This will save the user from having to log every transaction manually.

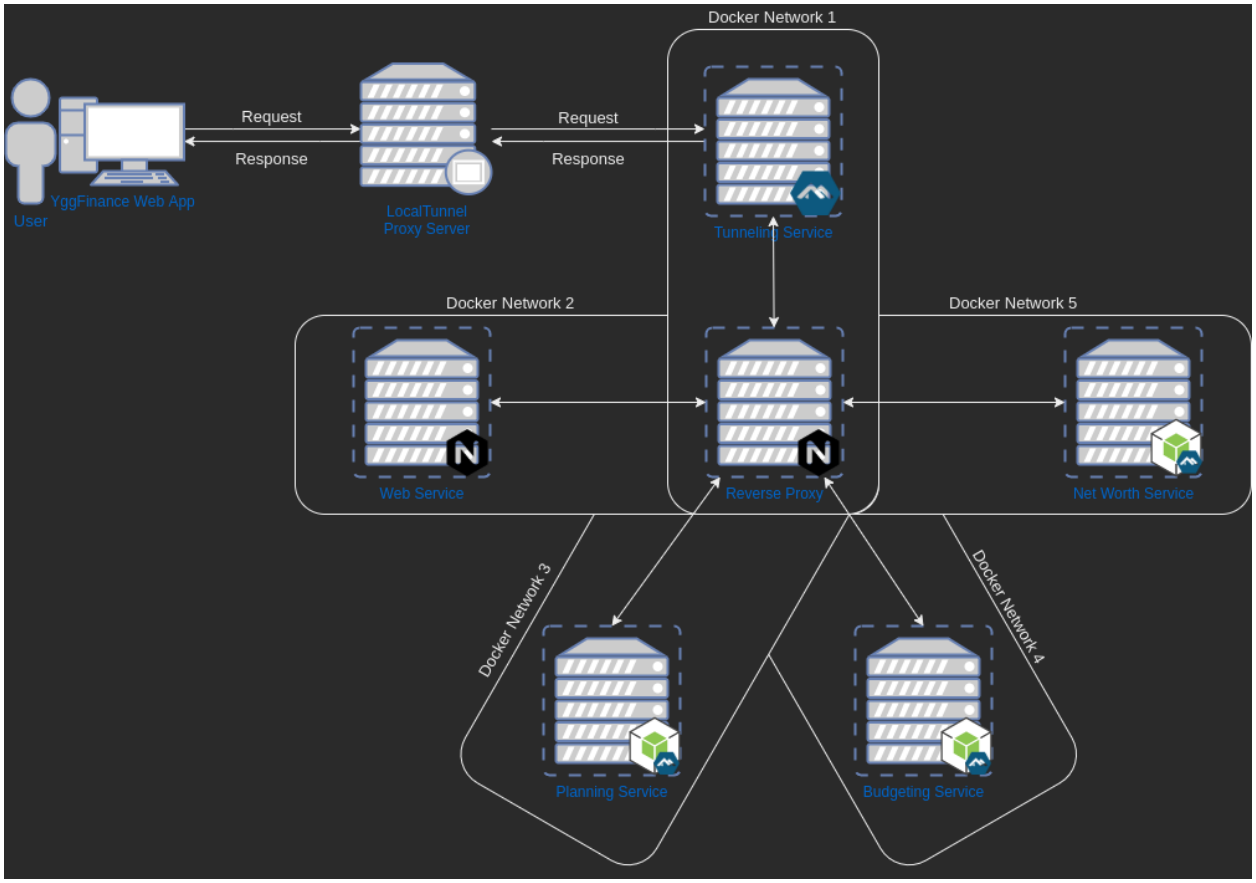
# Appendix

Please see below for the various tables, diagrams, screenshots, figures, etc. made throughout the document.

## A.1 Module Table

| Module | Category | Programming Language | Endpoint | Docker |
|---|---|---|---|---|
| YggFinance WebApp | Consumer | JavaScript | / | nginx:alpine |
| Reverse Proxy | Support Tool | N/A | N/A | nginx:alpine |
| Tunneling Utility | Support Tool | Bash | N/A | node:14-alpine |
| Budgeting Service | Service | JavaScript | /budgeting-service | node:14-alpine |
| Planning Service | Service | JavaScript | /planning-service | node:14-alpine |
| Net Worth Service | Service | JavaScript | /networth-service | node:14-alpine |

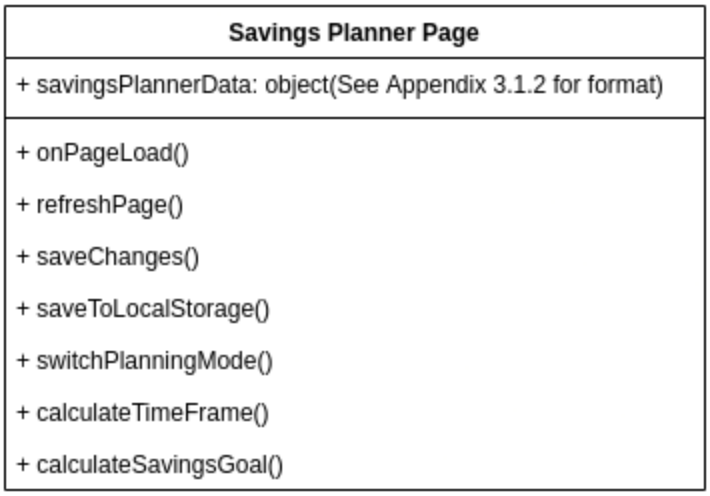
## A.2 System Topography Diagram



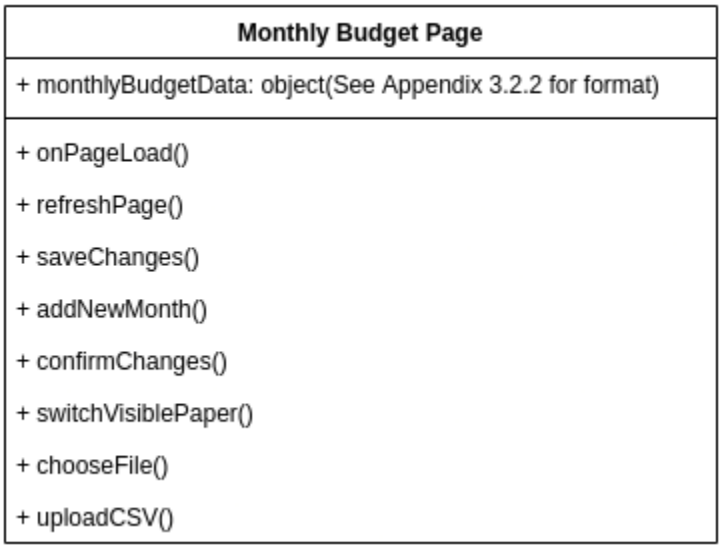

## A.3 UML Diagrams

This section contains the various UML diagrams referenced throughout this document

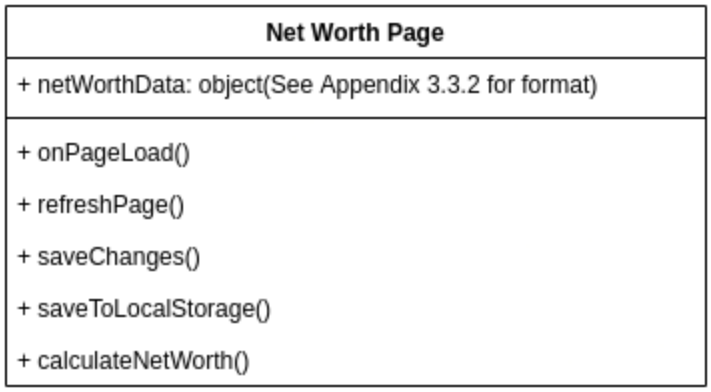
### A.3.1 Savings Planner Page UML Diagram



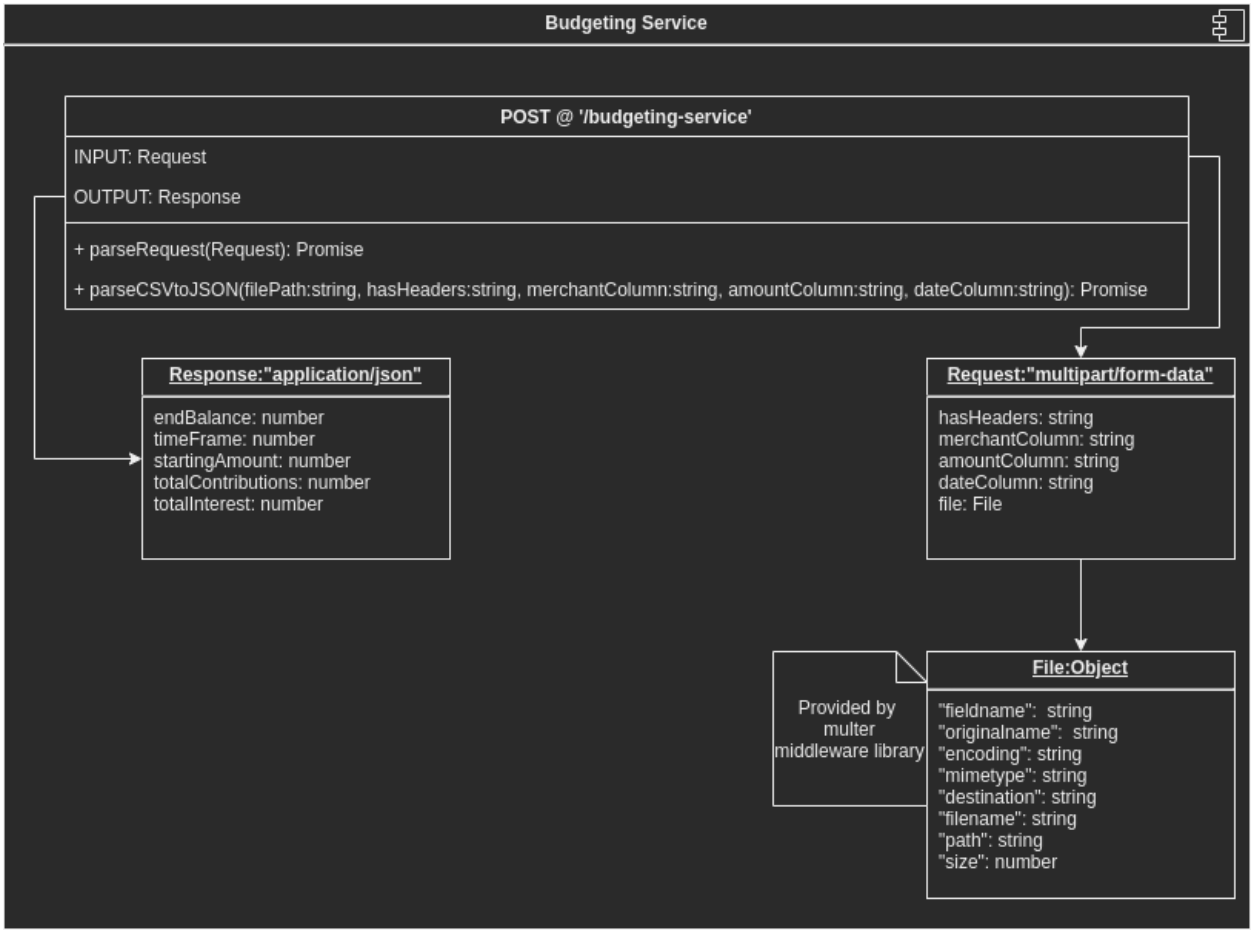

## A.3.2 Monthly Budget Page UML Diagram



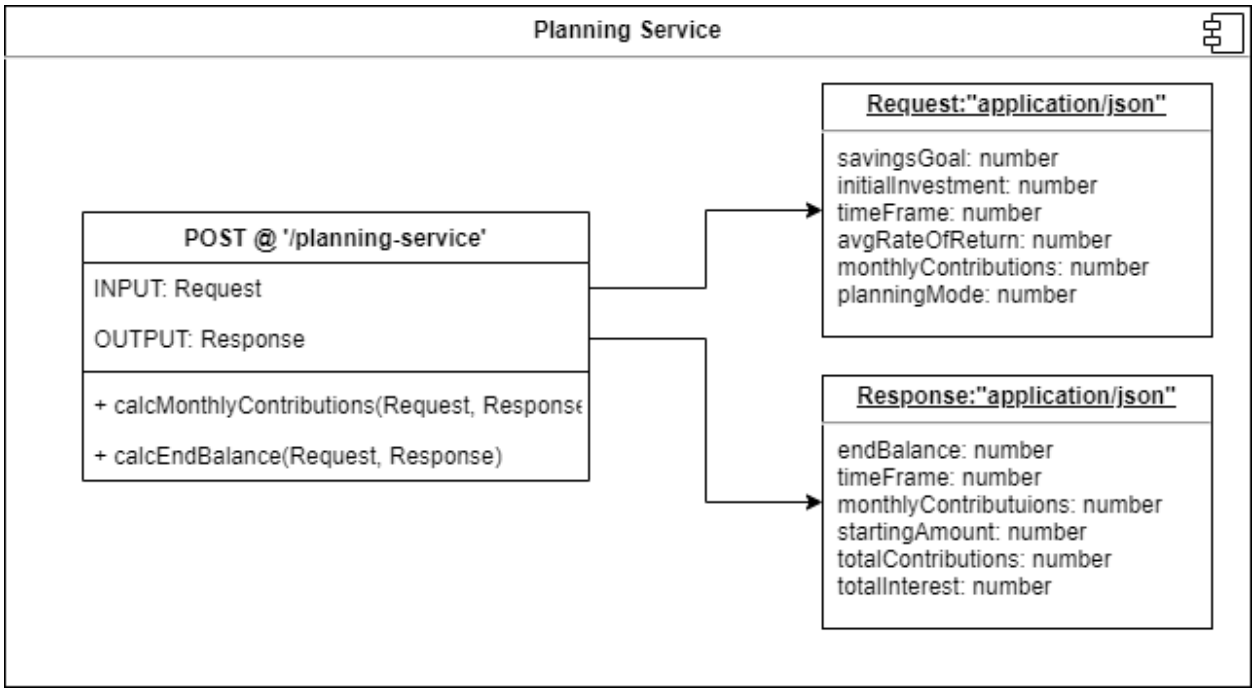

## A.3.3 Net Worth Page UML Diagram



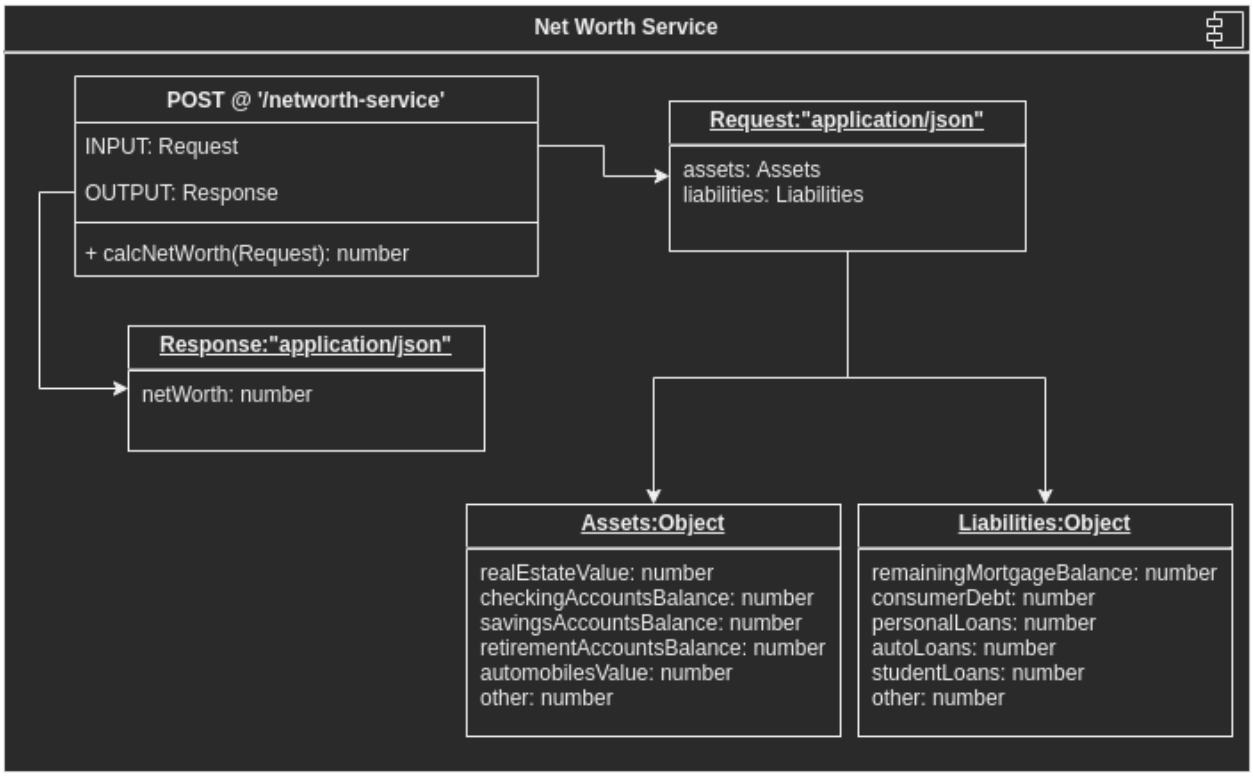

## A.3.4 Budgeting Service UML Diagram




## A.3.5 Planning Service UML Diagram



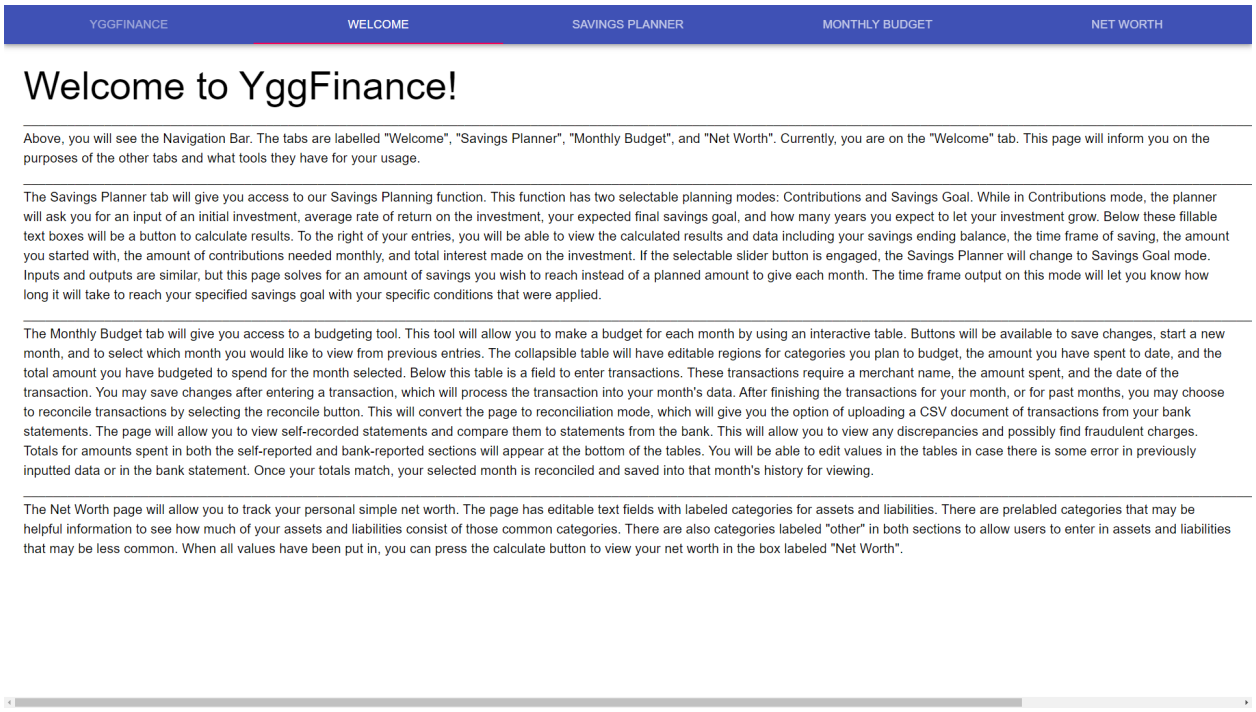

## A.3.6 Net Worth Service UML Diagram



# A.4 UI Screenshots

This section contains the various screenshots referenced throughout this document

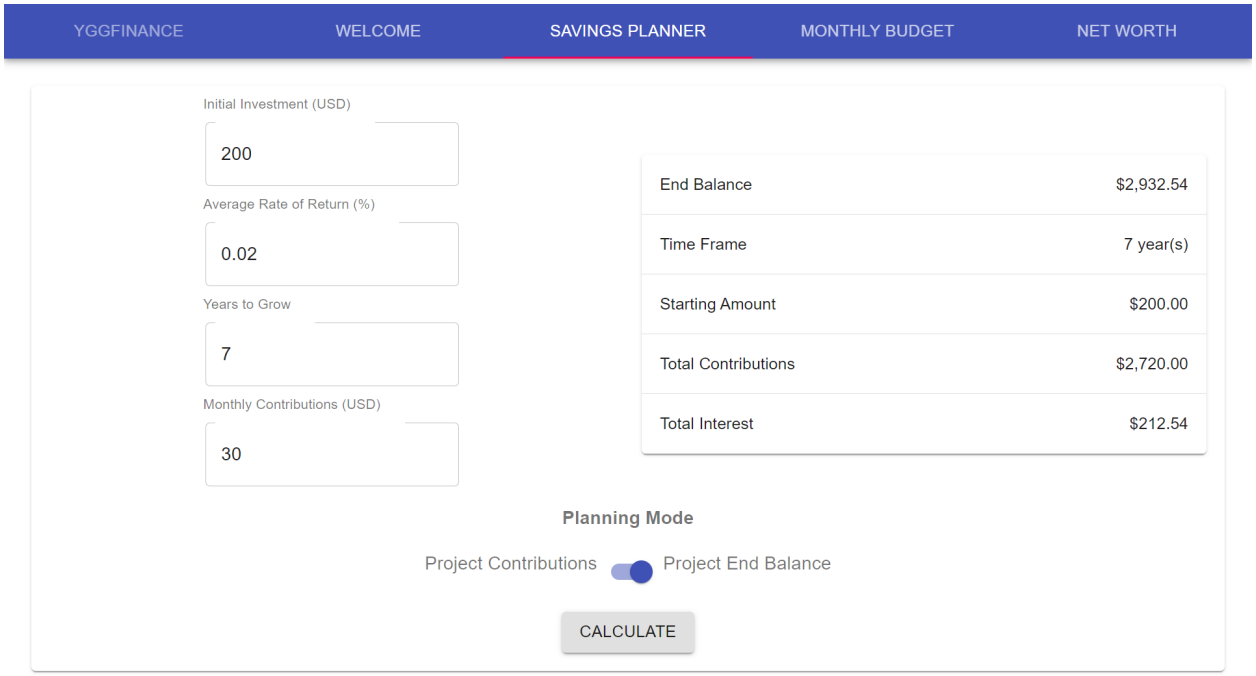
## A.4.1 Screenshot of Welcome Page UI



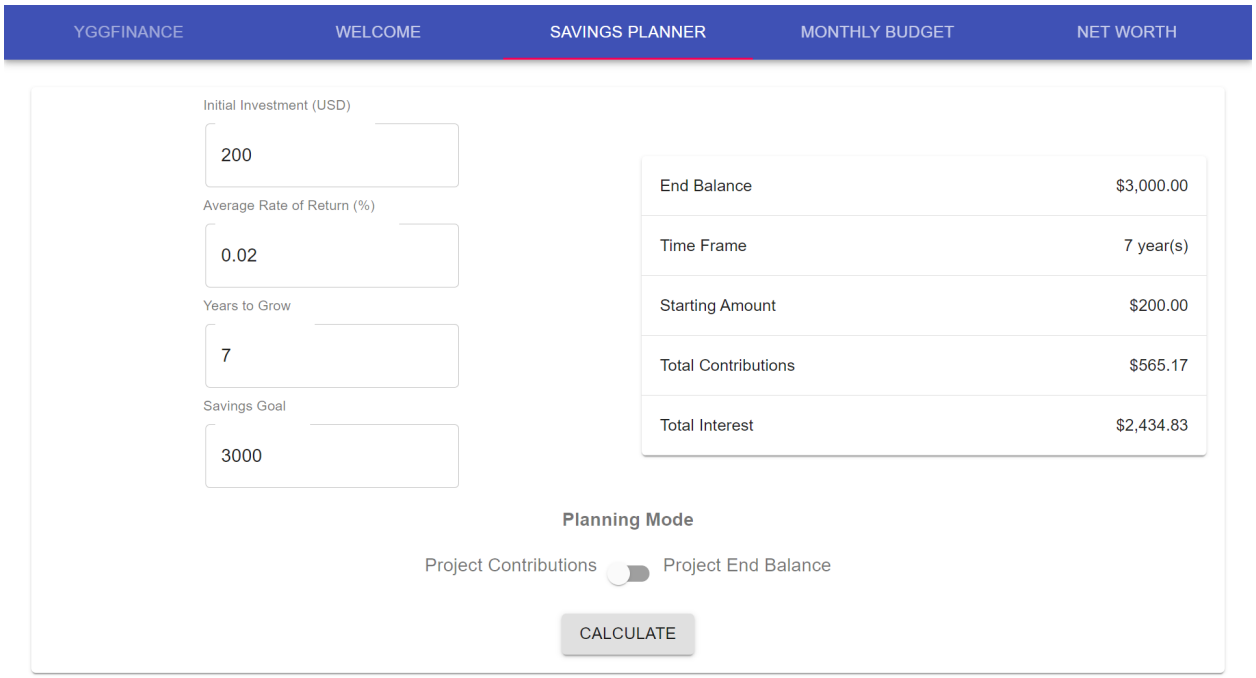

## A.4.2 Savings Planner Page UI

### A.4.2.1 Screenshot of Contribution Mode

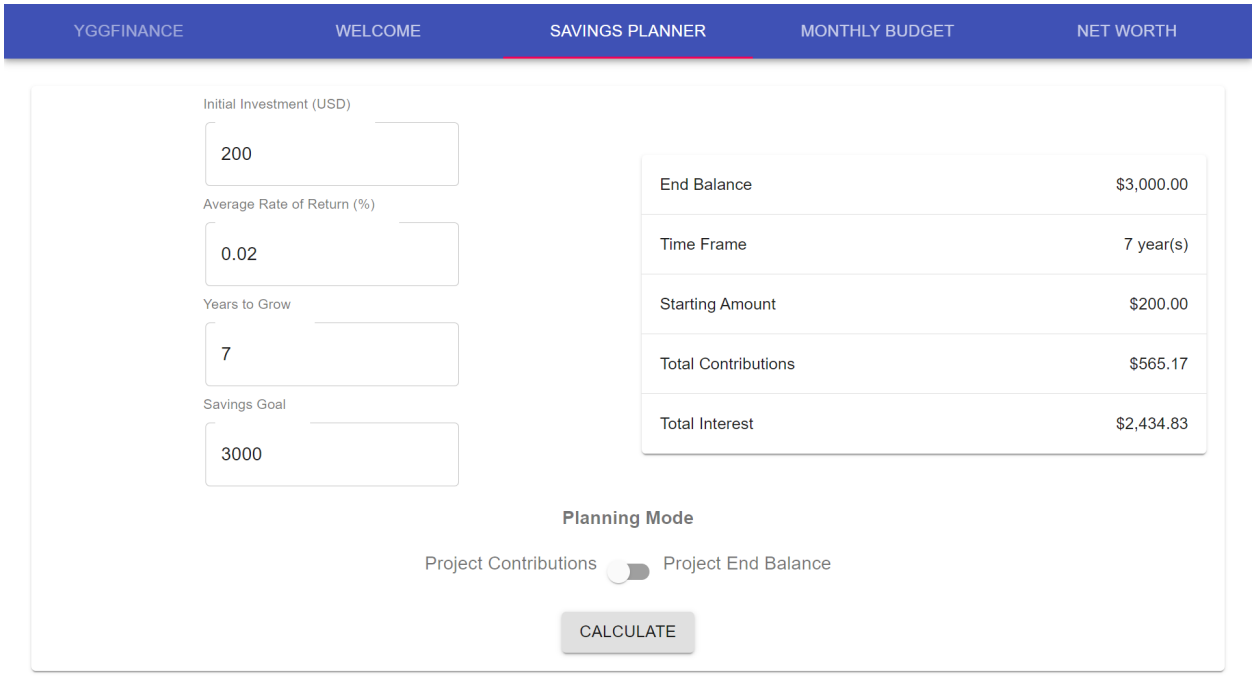
### A.4.2.2 Screenshot of Savings Goal Mode




## A.4.3 Monthly Budget Page UI

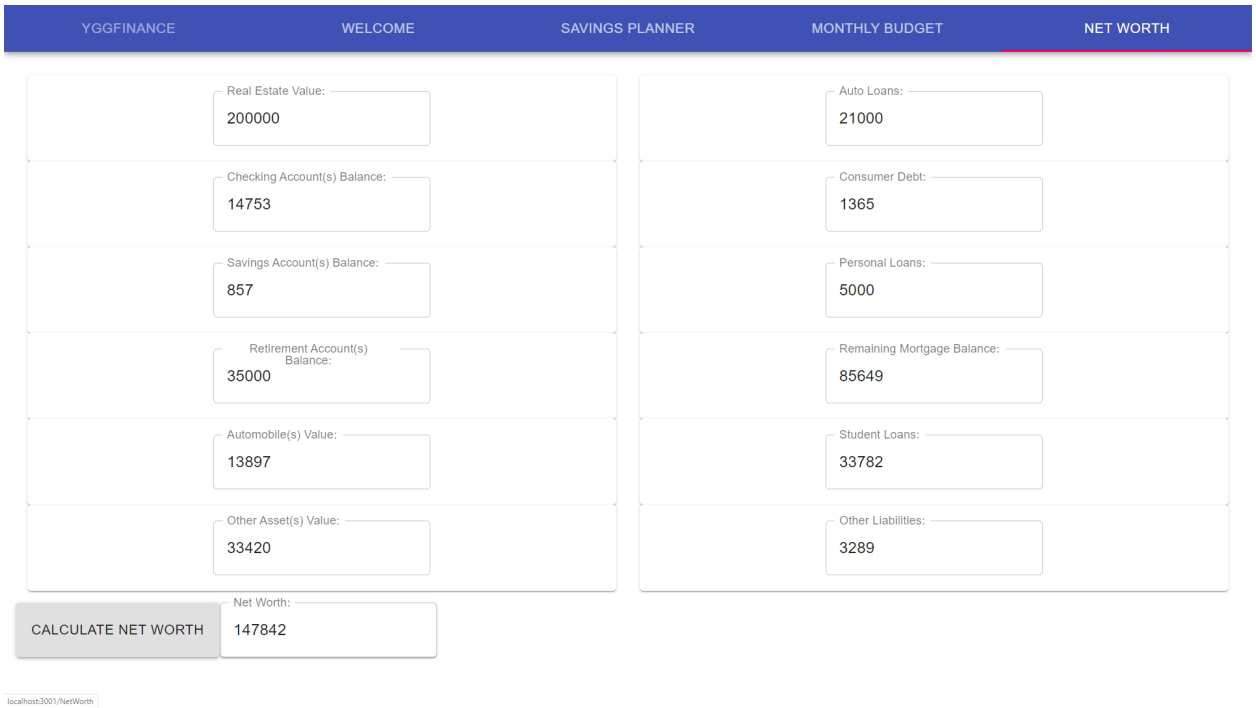



## A.4.3.1 Screenshot of Normal Mode




## A.4.3.2 Screenshot of Reconciliation Mode

## A.4.4 Screenshot of Net Worth Page UI



## A.5 LocalStorage

This section of the Appendix contains the localStorage JSON schema for YggFinance.

### A.5.1 Monthly Budget Page

```
const State = {
 selectedMonth: BudgetMonth,
 budgetedMonths : array[BudgetMonth],
 categories: array[Category],
 lastCategoryID: number,
 transactions: [Transaction],
 lastTransactionID: number
}

const BudgetMonth = {
 name: string,
 categories: array[ID:number]
}

const Category = {
 ID: number,
 name: string,
 transactions: array[ID:number],
 budget: number
}

const Transaction = {
 ID: number,
 merchant: string,
 amount: number,
 date: string, // (date.toJSON)
 isReconciled: boolean
}
```

### A.5.2 Savings Planner Page

```
const State = {
 initialInvestment: number,
```

```
 avgRateOfReturn: number,
 monthlyContributions: number,
 planningMode: boolean,
 timeFrame: number,
 savingsGoal: number
}
```

### A.5.3 Net Worth Planner Page

```
const State = {
 assets: Assets,
 liabilities: Liabilities
}

const Assets = {
 realEstateValue: number,
 checkingAccountsBalance: number,
 savingsAccountsBalance: number,
 retirementAccountsBalance: number,
 automobilesValue: number,
 other: number
}

const Liabilities = {
 remainingMortgageBalance: number,
 consumerDebt: number,
 personalLoans: number,
 autoLoans: number,
 studentLoans: number,
 other: number
}
```