Project: YggFinance

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

Table of Contents

Table of Contents **1**

1 Introduction **2**

2 Technology **3**

2.1 Architecture 3
2.2 Version Management 3
2.3 Programming Languages 4
2.4 NPM Dependencies 4
2.5 Base Docker Images 5
2.6 Scripting & Configuration 6
2.7 Application Persistence 6
2.8 Communication Standards 6
2.9 Proxy Tunneling 7
2.10 Development Tools 7
2.11 Documentation Tools 8
2.12 Team Communication 8

3 Design **8**

3.1 User Interface Design 8
3.1.1 Welcome Page 9
3.1.2 Saving Planner Page 9
3.1.3 Monthly Budgeting Page 9
3.1.4 Net Worth Page 9
3.1.5 Database Tables (LocalStorage) 9
3.1.6 Script Files 9
3.1.7 Required Data Files 9
3.2 Service Design 10
3.2.1 Budgeting Service 10
3.2.2 Planning Service 10
3.2.3 Net Worth Service 10
3.2.4 Web App (backend) 11

3.2.5 Reverse Proxy 11
3.2.6 Tunneling Service 11
3.2.7 Database Tables (None) 11
3.2.8 Script Files 11
3.2.9 Required Data Files 11

4 Deploying 11

5 Building 12

6 Known Bugs 13

7 Future Work 13

Appendix 13
A.1 Module Table 13
A.2 Communication Topography Diagram 14
A.3 UML Diagrams 14
A.3.1 Savings Planner Page UML Diagram 14
A.3.2 Monthly Budget Page UML Diagram 15
A.3.3 Net Worth Page UML Diagram 15
A.3.4 Budgeting Service UML Diagram 16
A.3.5 Planning Service UML Diagram 16
A.3.6 Net Worth Service UML Diagram 17
A.4 UI Screenshots 17
A.4.1 Screenshot of <XYZ UI Page thing> 18

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. Ygg Finance assists users in setting goals by providing a calculator to look at future projections of their investments allowing them to tweak the numbers until they find the correct amount to save. 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 very broad picture 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 both far outside of our expertise and are 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 service application 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 Service and the **Nginx:alpine** Base Container Image for the Reverse Proxy & Web Service. 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 service 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

TODO: Explain the overall structure of our application while referencing figures from the reference sections following

3.1 User Interface Design

1. A **very** technical description of the design. You should list and describe
    1. All classes

    2. All database tables
    3. All script files
    4. All necessary data files
2. When describing the design, you should include whatever makes your description clear. You can include, but are not limited to
    1. clear text descriptions
    2. supporting diagrams and tables
    3. screenshots of the application

TODO: Intro and Explain the overall frontend UI Design while referencing figures from the reference sections following

3.1.1 Welcome Page

TODO: Describe how the welcome page is designed while referencing figures from the reference sections following

3.1.2 Saving Planner Page

TODO: Describe how the savings planner page is designed while referencing figures from the reference sections following

3.1.3 Monthly Budgeting Page

TODO: Describe how the monthly budgeting page is designed while referencing figures from the reference sections following

3.1.4 Net Worth Page

TODO: Describe how the net worth page is designed while referencing figures from the reference sections following

3.1.5 Database Tables (LocalStorage)

TODO: Create a ERD/UML diagram for the data stored in LocalStorage

3.1.6 Script Files

TODO: Talk about the use of create-react-app and the scripts it employs

3.1.7 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

1. A **very** technical description of the design. You should list and describe
   a. All classes (Modules/Services)
   b. All database tables (None)
   c. All script files
   d. All necessary data files
2. When describing the design, you should include whatever makes your description clear. You can include, but are not limited to
   a. clear text descriptions
   b. supporting diagrams and tables
   c. screenshots of the application

TODO: Intro and Explain the overall backend Service Design while referencing figures from the reference sections following

3.2.1 Budgeting Service

TODO: Describe how the budgeting service is designed while referencing figures from the reference sections following

(See A.3.4 Budgeting Service UML Diagram)

3.2.2 Planning Service

TODO: Describe how the planning service is designed while referencing figures from the reference sections following

(See A.3.5 Planning Service UML Diagram)

3.2.3 Net Worth Service

TODO: Describe how the net worth service is designed while referencing figures from the reference sections following

(See A.3.6 Net Worth Service UML Diagram)

3.2.4 Web App (backend)

TODO: Describe how the web app backend serves the client WebApp content to the client.

3.2.5 Reverse Proxy

TODO: Describe how the reverse proxy is designed while referencing figures from the reference sections following

3.2.6 Tunneling Service

TODO: Describe how the tunneling service is designed while referencing figures from the reference sections following

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 Script Files

TODO: Mention and explain the make-release script file

3.2.9 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 service 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-service 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

1. Your system probably has bugs that you are aware of but have run out of time and cannot fix. Explain each known bug and provide an explanation of the approach the team will take to fix the bug for the next release.

TODO: Write the known bugs in our application (this could even mean literal features that are mentioned in this document not working btw)

7 Future Work

1. Now that the version 1 is finished, provide a list of features that will be added in the version 2. Write this as if version 2 will be released in four months.

TODO: Write about our planned future work here

Appendix

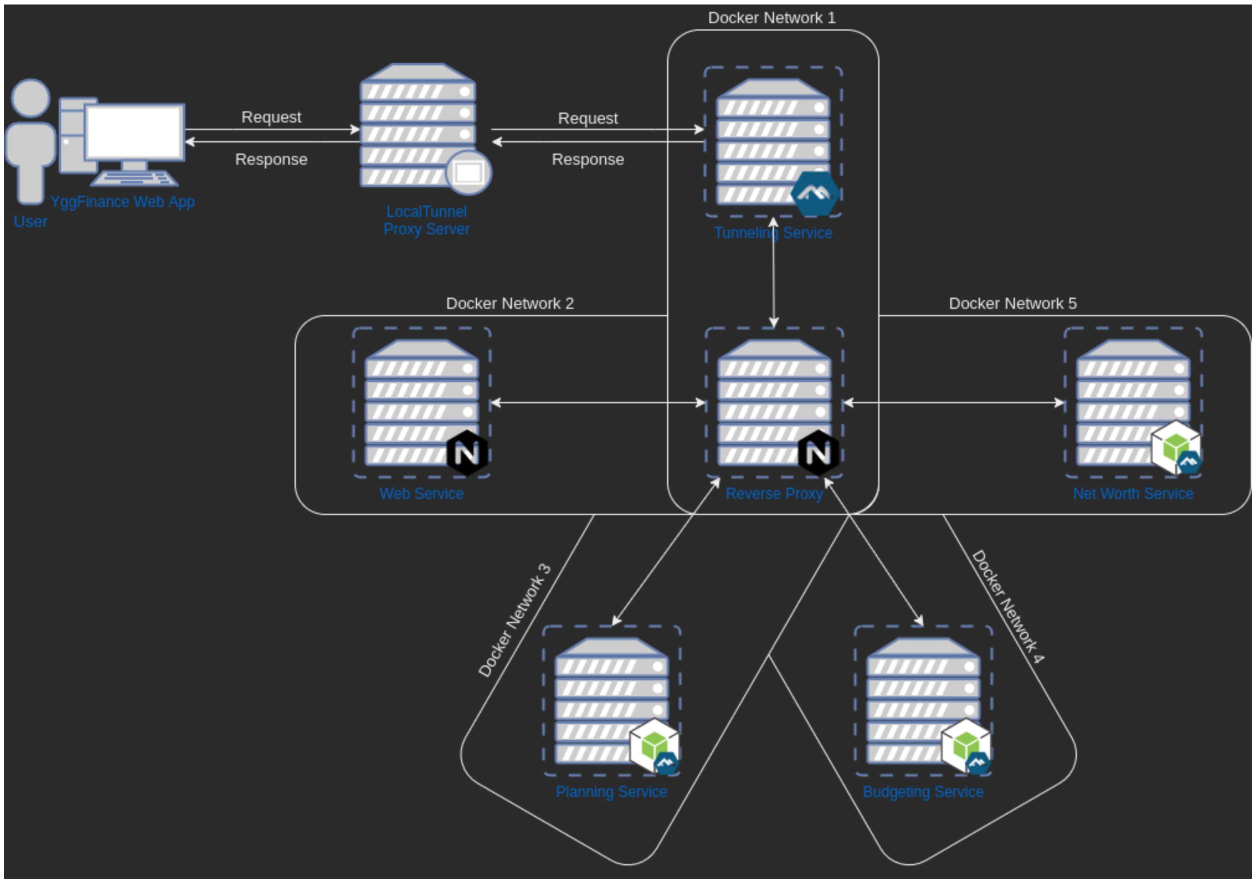
Please see below for the various tables, diagrams, screenshots, figures, etc. made throughout the document.

A.1 Module Table

Module	Category	Programming Language	Sdk	Docker
YggFinance WebApp	Consumer	JavaScript	Node	nginx:alpine
Reverse Proxy	Support Tool	-	-	nginx:alpine

Tunneling Service	Support Tool	Bash	-	node:14-alpine
Budgeting Service	Service	JavaScript	Node	node:14-alpine
Planning Service	Service	JavaScript	Node	node:14-alpine
Net Worth Service	Service	JavaScript	Node	node:14-alpine

A.2 Communication Topography Diagram

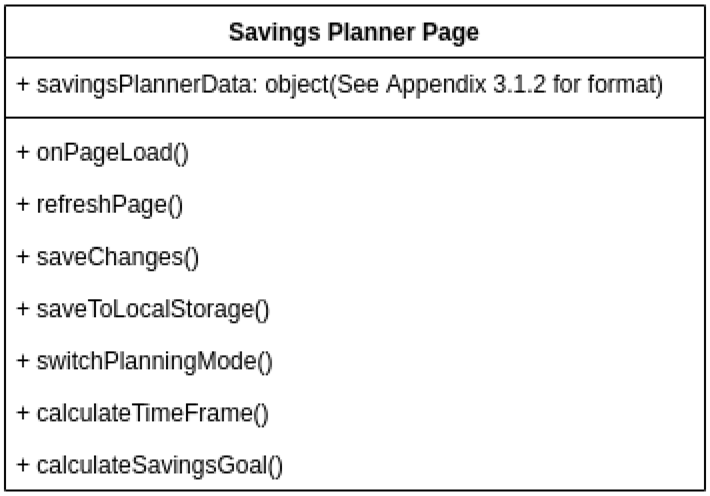


A.3 UML Diagrams

This section contains the various UML diagrams referenced throughout this document

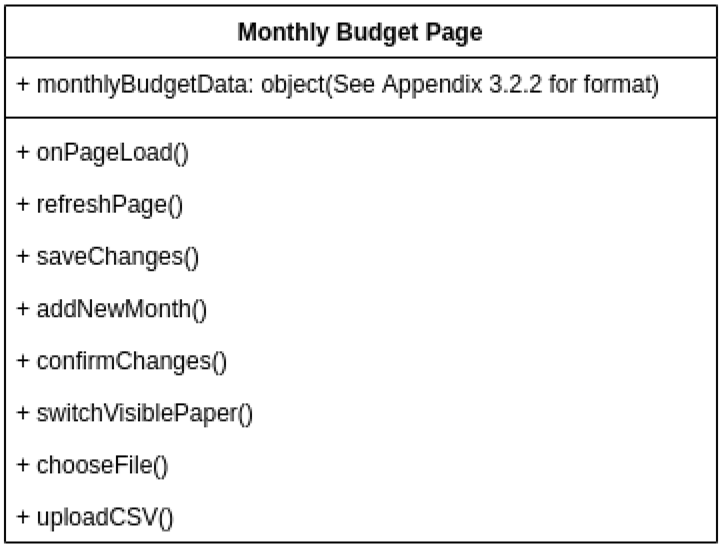
A.3.1 Savings Planner Page UML Diagram

TODO: Update the UML Diagrams for the UI and post here



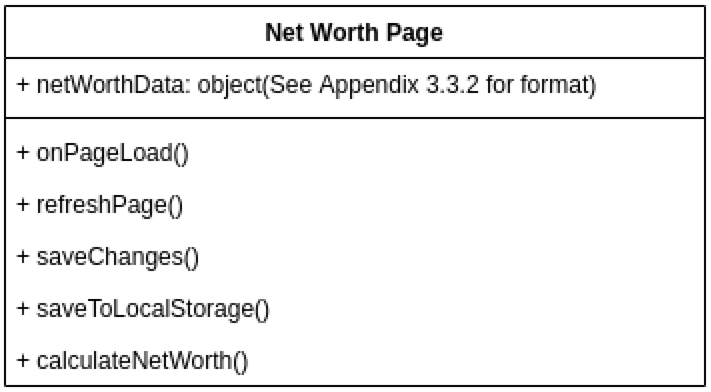
A.3.2 Monthly Budget Page UML Diagram

TODO: Update the UML Diagrams for the UI and post here

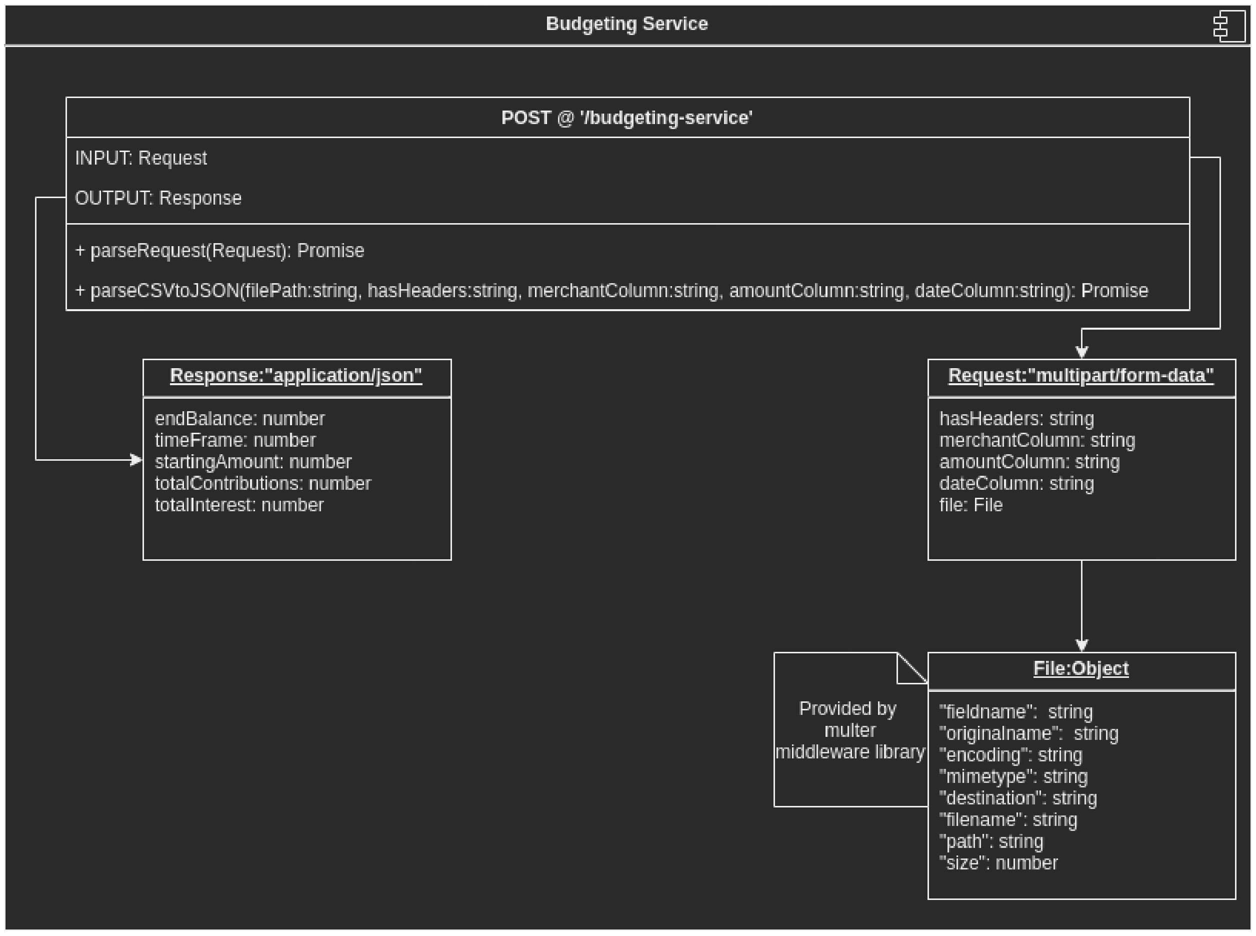


A.3.3 Net Worth Page UML Diagram

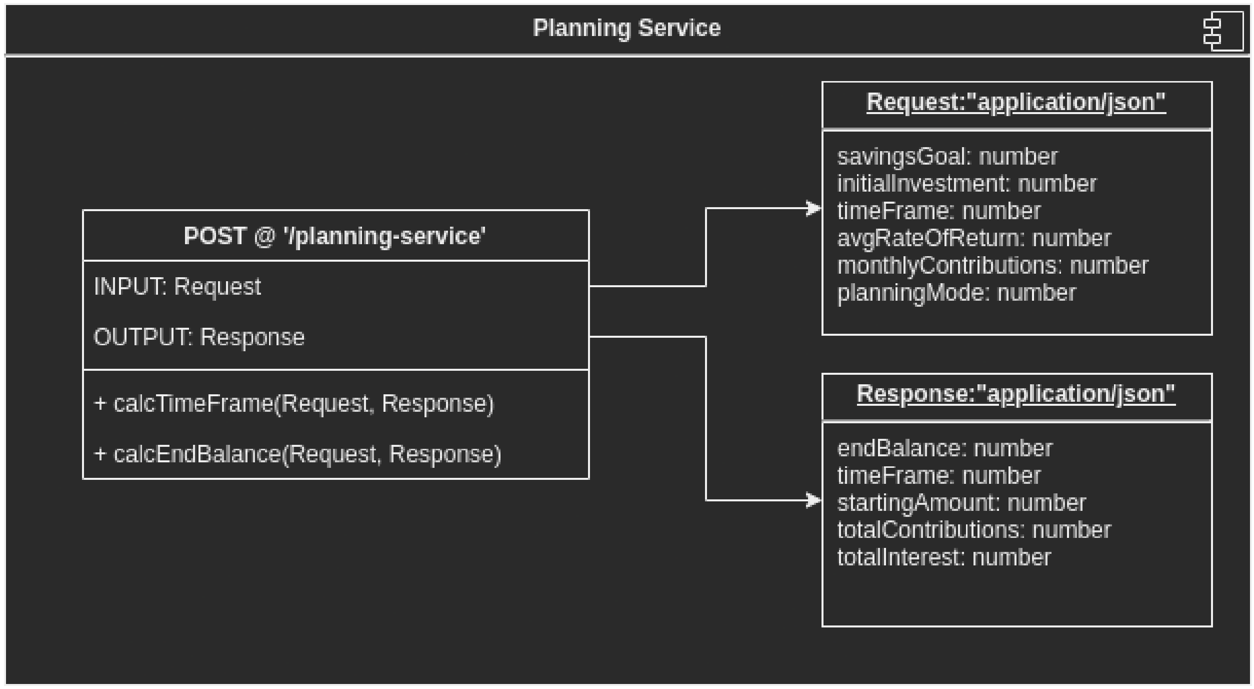
TODO: Update the UML Diagrams for the UI and post here



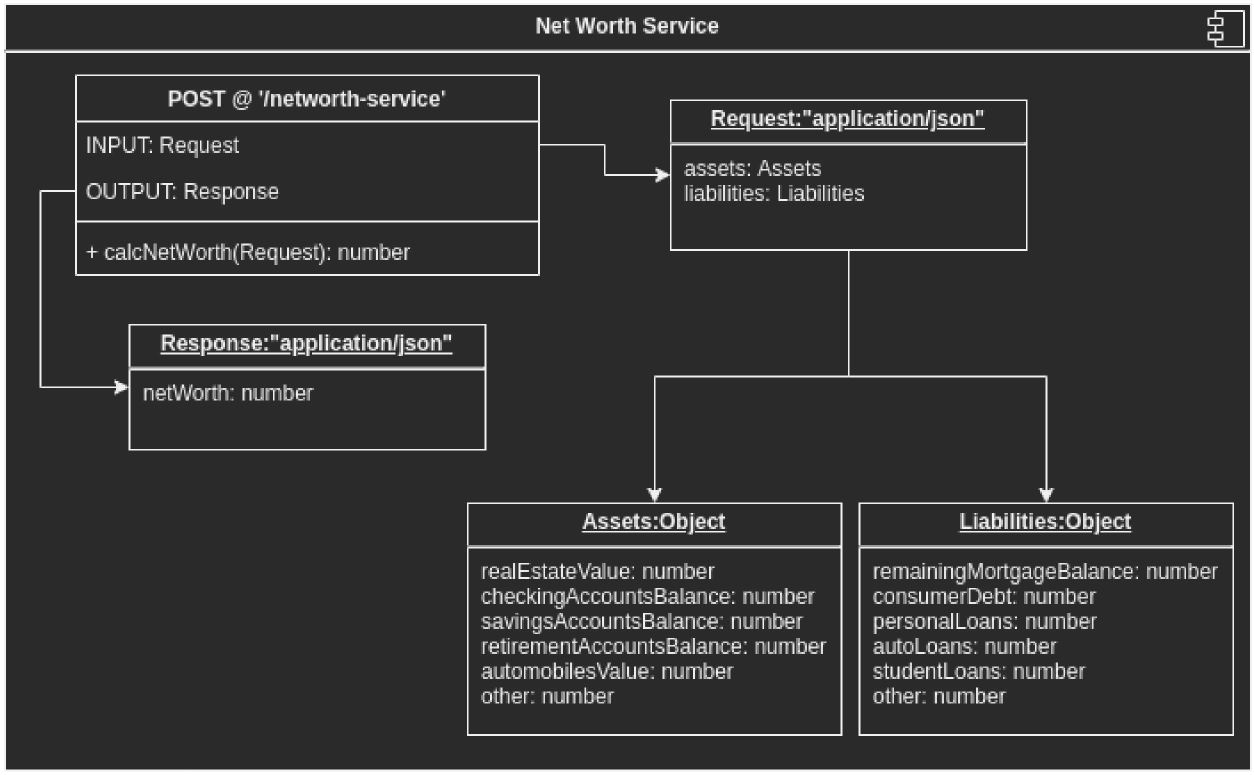
A.3.4 Budgeting Service UML Diagram



A.3.5 Planning Service UML Diagram


Planning Service
POST @ '/planning-service'
INPUT: Request
OUTPUT: Response
+ calcTimeFrame(Request, Response)
+ calcEndBalance(Request, Response)
Request:"application/json"
savingsGoal: number
initialInvestment: number
timeFrame: number
avgRateOfReturn: number
monthlyContributions: number
planningMode: number
Response:"application/json"
endBalance: number
timeFrame: number
startingAmount: number
totalContributions: number
totalInterest: number

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 <XYZ UI Page thing>

TODO: Take screenshots of the Web App for reference