Home Financial Planning Infrastructure

SpendLyte

Final Report (Semester Two)

Joe Wemyss

20068336 Panel 14

Supervisor: Catherine Fitzpatrick

BSc. (Hons) Software Systems Development

Table of Contents

Index of Tables

Project URL: https://spendlyte.com

Introduction

The overarching goal of this project is to provide a comprehensive service that allows users to manage their household resources and processes.

This is a very broad, high-level overview of what this project aims to achieve, and needs to be disseminated to determine exactly what is meant by this statement.

> The overarching goal of this project is to provide a **comprehensive** service that allows users to manage their household resources and processes.

One metric for success is how comprehensive the finished project will be. The purpose the of this service is to build an infrastructure that could be used to manage all aspects of a household. This means that the system should be able to integrate with other applications that provide similar services.

> The overarching goal of this project is to provide a comprehensive service that allows **users** to manage their household resources and processes.

Another metric of success for this project would be how user-centric it is. The client facing portion of the application should be designed with simplicity in mind to allow the system to be used by people of all technical skill levels. The system should also be designed to work across as many platforms as possible.

> The overarching goal of this project is to provide a comprehensive service that allows users to manage their household **resources** and processes.

Resources is a very broad term. Within the context of this project, it refers to the resources used to run a home. This could range from money to commodities (such as cleaning products, toilet paper) to utilities (gas, electricity, phone, internet) to food and beyond. As can be seen from this list, financial planning is going to be the core of the application. Practically all resources used by a household must be procured using money. For this reason, the financial planning section of the application will need to be focused on more than the rest.

> The overarching goal of this project is to provide an easy to use service that allows users to manage their household resources and **processes**.

Processes is another very broad term. In this case, it refers to the processes that take place in everyday life, somewhat analogous to business processes in enterprise. An example of this would be procuring the food for the household or delegating chores to members of the household. Therefore, the system must be flexible enough to manage processes for many households, as well as dynamic enough to change to match user requirements.

Project Outline

Realistically, this application should incorporate a fully featured client and server, but due to time constraints, this will not be possible. Instead it was decided that the main focus would be on the server side functionality. The reason that this was decided is that the server-side infrastructure is the most important aspect of the application, since it is intended to be developed upon by third party developers.

An example of a third party integrating with this application would be, for example, a pre-pay electricity company building an application that would allow their meter to send periodic updates about electricity usage to the system, or a supermarket building an app that allows users to scan receipts to have the data automatically uploaded.

This application is also designed to be a simple prototype to showcase some of what this application could do in the future. As such it is not intended to carry the full suite of planned features.

Technology Overview

Database

The database that is used in this system is Firebase Cloud Firestore. This is an offering from Google which fits into the Firebase suite. Cloud Firestore is designed to be accessed from either the client or the server, which was one of my major reasons for adopting it. I felt that this would allow me to complete easy operations in the directly in the client without the need for making an API call to the server.

Firestore also allows for data to be transparently available offline. This means that users will be able to access their data even when they do not have internet access.

Server

In this case the server is a set of Firebase Cloud Functions. These are another Google offering that can be used as part of the Firebase suite. The advantage of using Cloud Functions over a standard REST API is that the server can be converted into a set of “apps” which share a common codebase and can be scaled independently. This offers more simplicity than micro-services, as they can all be built using a single codebase with different sets of route files, and more scalability than a traditional REST API as each “app” scales automatically based on the load currently hitting its endpoints.

The actual technologies used to implement the server on top of Firebase are NodeJS, Express and TypeScript. NodeJS is a runtime built on top of Googles V8 engine that allows JavaScript to be run on the server. Node works particularly well for handling typical HTTP/s communications as its event loop can handle extremely large volumes of concurrent requests without the need for any multi-threaded implementation. This is also its major downfall, as it can never be multi-threaded, to scale a new process needs to be started. This is not an issue for REST API’s however as they are stateless, so there is no need for inter-process communication.

Express is a framework for building applications in NodeJS. Express greatly simplifies the process of implementing a web application, as it provides a very high level abstraction from the underlying mechanisms. An Express application is made up of sets of routes and middlewares. Each request will be passed through a series of middlewares before it reaches the appropriate route handler. This project is made up of three separate Express applications that scale independently.

TypeScript is a superset of JavaScript which was created by Microsoft. It allows for static type checking at runtime and compiles to standard JavaScript to be executed by the Node runtime. TypeScript also has the benefit of being ES+ compliant, meaning that the latest features will be available to use even if they have not been implemented yet. For example, the new *for-await-of* functionality was only confirmed as being part of the ECMAScript 2018 specification in January, but already exists in TypeScript 2.6.

Full Technology List

This is the full list of technologies that are used in the server-side implementation of the system. The server-side dependencies are not as important to manage as the client-side dependencies, as they will not have to be sent over the wire when a client connects, and as such some of the dependencies listed below are related to legacy features which have not been pruned.

Dependencies

These are the actual dependencies required by the application at run-time. These dependencies ensure that the application is able to function.

Name	Version	Description
Cookie-Parser	1.4.3	Express plugin for parsing incoming cookies
Cors	2.8.4	Express plugin for easy setup of 'Access-Control-Allow-Origin' headers to prevent 'Cross-Origin-Resource-Sharing' issues
Date-fns	1.29.0	Functional style date utilities
Errorhandler	1.5.0	Development Error handling and stack traces
Express	4.16.2	Application Framework for NodeJS
Firebase-Admin	5.8.1	Server-side API for interacting with Firebase services
Method-Override	2.3.10	Allow the use of HTTP verbs where the client does not support them
Moment	2.20.1	Object-Oriented date utils

Development Dependencies

These are the dependencies required by the application at compile time. These dependencies ensure that the application can be built, and also streamline the development process.

Name	Version	Description
Chai	4.1.2	Assertion framework for testing
Cross-Env	5.1.3	Environment agnostic way of setting environment variables in an NPM script
Mocha	5.0.0	JavaScript test runner
Nodemon	1.14.12	Watch files and restart Node server on changes
Sinon	4.2.2	Stubbing/Mocking library for testing
Sinon-Chai	2.14.0	Chai plugin for sinon assertions
TS-Node	4.1.0	Allow Node to run TypeScript files without first compiling to JavaScript
TSLint	5.8.0	Linting for TypeScript files
TypeScript	2.5.3	Super-set of strongly type, ES+ JavaScript

Client

For this project I chose VueJS as my client side framework. This framework allows for users to create applications using JavaScript in the browser. The reason that I chose Vue over more mature offerings, such as AngularJS, Angular and React, is that it combines the performance of Reacts virtual DOM and the ability to use HTML style templates from Angular/JS. I also like the Single File Component layout of Vue components.

As my client side style framework I chose Vuetify. The reason for this is that Vuetify implements Googles Material Design specification, which is a well known and aesthetically pleasant design. Vuetify also provides a very large collection of pre-built components that can be implemented very easily. These components can be then combined to make larger and more intricate UI's. I also chose Vuetify for security. Vuetify does not use dangerous operations (such as *element.innerHTML)* meaning that it is less vulnerable to Cross-Site Scripting (XSS) attacks.

Full Technology List

Dependencies

This is quite possibly the most important dependency list in the application. The reason for this is that each of these modules will be bundled into a single webpack chunk to be sent to the client. This chunk is by far the largest resource that must be loaded by the client, and as such must be optimised as much as is possible.

Name	Version	Description
Axios	0.17.1	HTTP client for making AJAX requests
Date-fns	1.29.0	Functional style date utilities
Firebase	4.10.0	Client-side library for interacting with Firebase services
Localforage	1.5.5	Abstraction over localStorage/WebSQL/IndexdDB implementation
Vee-Validate	2.0.4	Client side validation library for validating form fields
Vue	2.5.3	Client side application framework
Vue-Currency-Filter	2.1.0	Filter for formatting numbers as currency
Vue-Router	3.0.1	In app routing for vue applications
Vuetify	1.0.4	Material Design components for Vue
Vuex	3.0.1	State management for Vue
Vuex-Router-Sync	5.0.0	Sync route changes to Vuex store

Development Dependencies

For the development dependencies on the client, I have not listed them all. This is because are a total of 80 development dependencies, at least 12 of which are directly related to babel and at least 27 are directly related to webpack. In the interest of brevity, only the more important modules are listed here.

Name	Version	Description
Autoprefixer	7.1.2	Vendor CSS prefixing for browser compatibility
Babel	6.22.1	Transpilation of JavaScript to lower ES versions
Karma	2.0.0	Test runner
Flow-Bin	0.61.0	Static type checking for JavaScript
Babel-Plugin-Istanbul	4.1.1	Code coverage plugin for babel
TestCafe	0.18.5	E2E test runner
Webpack	2.6.1	JavaScript module bundler
TSLint	5.8.0	Linting for TypeScript files
TypeScript	2.5.3	Super-set of strongly type, ES+ JavaScript

Data Structure

Since this project is mainly server-side focused, the most complex data models are found in the server side. These models are used to map the data from its physical representation in the database, to a representation that can be viewed by the client. This was the most intensive segment of building the application.

Database

As the data is stored in Firebase Cloud Firestore, it is represented in JSON format. More specifically, it is a key/value datastore. This means that instead of having collections of documents (in the manner of MongoDB), the entire database is stored in a single JSON tree graph consisting of nodes.

The structure of the database consists of three root nodes. These nodes are *items, tags* and *transactions.* Underneath each root node, there are nodes with

Tags

The tags node in the database is used to track tags that are used by users. The reason for keeping these tags in a separate node is that they can then be used for auto-completion in the client side application. This node differs from the others in that it has a *public* node which exists under the root. This node contains some default tags that will be available as part of the auto-complete for all users. Then custom tags are added to the node specified by the users id. These tags are added automatically when a user adds an item to the database which contains new tags. The tags are stored in JSON format, rather than as an array of strings, due to a nuance of the Firestore implementation which does not allow single items to be added to an array, instead the entire array must be overwritten. By saving as JSON and using the name of the tag as a key, it is possible to insert new elements. A simple example of how this would look with two users on the systems is represented below:

```
"tags": {
  "public": {
    "expense": true,
    "home": true,
    "food": true,
    "income": true,
    "utilities": true
  },
  "It10AYay5gWqpT4100NDEr8Skyk2": {
    "joe": true,
    "jane": true
  },
  "Fm9jNcLltwMCLDPREEH0tNJZz283":{
    "mick": true,
    "pat": true,
    "butter": true,
    "entertainment": true
  }
}
```

Transactions

The transactions node is the original store for storing transactional data. It is used to store records of financial activity for users. Data is stored in a node which uses the users ID as a key. Under this there are two sub-collections, *incomes* and *expenditures.* Each transaction under these two nodes has its own unique ID. Each transaction simply contains an amount, a frequency that the transaction should occur, the next time the transaction is due and a title. The next time a transaction is due is combined with the frequency to generate lists of dates of when transactions will occur.

This method of storing data has now been deprecated and is no longer used. It is simply retained for backwards compatibility, showcasing some functionality in the client that has not been implemented with the new data model and showing the development of the applications underlying data structure. An example of this node in the same scenario is above is shown below:

```
  "transactions": {
    "It10AYay5gWqpT4100NDEr8Skyk2": {
      "incomes":{
        "14dBCkeY3gBFwvYXZl7I":{
          "amount": 2500,
          "frequency": "monthly",
          "nextDueDate": "2018-04-26",
          "title": "wages"
        }
      },
      "expenditures": {}
    },
    "Fm9jNcLltwMCLDPREEH0tNJZz283": {
      "incomes":{},
      "expenditures": {
        "fddBCkeY3gBFwvsdhijas":{
          "amount": 2.19,
          "frequency": "once",
          "nextDueDate": "2018-04-26",
          "title": "butter"
        }
      }}}
```

Items

The items datastore is by far the most comprehensive in the application. The purpose of this datastore is to replace the older and less detailed *transactions* datastore. The items datastore follows much the same pattern as the other two mentioned above, in that each users data is stored using their ID as a key.

This datastore stores mostly similar data to what is stored in the transactions datastore, exept that it is far more comprehensive and allows for much greater flexibility when generating dates. For example, the older method could only store frequency as either “daily”, “weekly” or “monthly”. The new datastore allows for more flexibility when selecting recurrence periods, such as setting a transaction to occur on the last Thursday of every second month. Another example of how it differs is that it implements the new tag system to store tags for each record. The new datastore also provides a mechanism to store past transactions to compare how the budgeted figure stacks up with the actual figure.

An example of this node in the same scenario is above is shown below:

```
  "items":{
    "It10AYay5gWqpT4100NDEr8Skyk2":{
      "records":{
        "peha9KZ97RyqGXCNWd0d":{
          "past":{
            "BZ9IqRr92KTL6SrUz950":{
              "budgeted": 2500,
              "actual": 2500,
              "completed": true,
              "date": "2018-02-22"
            },
            "dsfbrgdqJZfj1paj73T1":{
              "budgeted": 2500,
              "actual": 2500,
              "completed": true,
              "date": "2018-03-29"
            }
          },
          "amount": 2500,
          "dates": {
            "dates": ["2018-04-26"],
            "freq01": "Last",
            "freq02": "Thursday",
            "freqType": "wdim",
            "frequency": "MONTHLY",
            "interval": 1
          },
          "isIncome": true,
          "tags": ["income", "wages"],
          "title": "wages"
        }
      },
      "Fm9jNcLltwMCLDPREEH0tNJZz283": {
        "records": {
          "GhKwQh7GYGzMWGHSXvae": {
            "past":{
              "NLHOlH6bG8zWGs0YzGOY":{
                "budgeted": 2.19,
                "actual": 2.25,
                "completed": true,
                "date": "2018-04-20"
              },
              "vijY3jdgawArtWcCwehv":{
                "budgeted": 2.19,
                "actual": 2,
                "completed": true,
                "date": "2018-04-16"
              }
            },
            "amount": 2.19,
            "dates": {
              "dates": [
                "2018-04-26"
              ]
            },
            "isIncome": false,
            "tags": [
              "expense",
              "food",
              "household"
            ],
            "title": "butter"
          }
        }
      }
    }
  }
```

Server

The server data model is the most complex of the three, as this is where all of the calculations and aggregations will be performed before being sent on to the client. This allows as much functionality as possible to be kept on the server, making it easier for third parties to integrate. This also reduces the amount of redundant data that is stored in the database, making for easier maintainence and reduced storage costs. A simple overview of the data model is represented below.

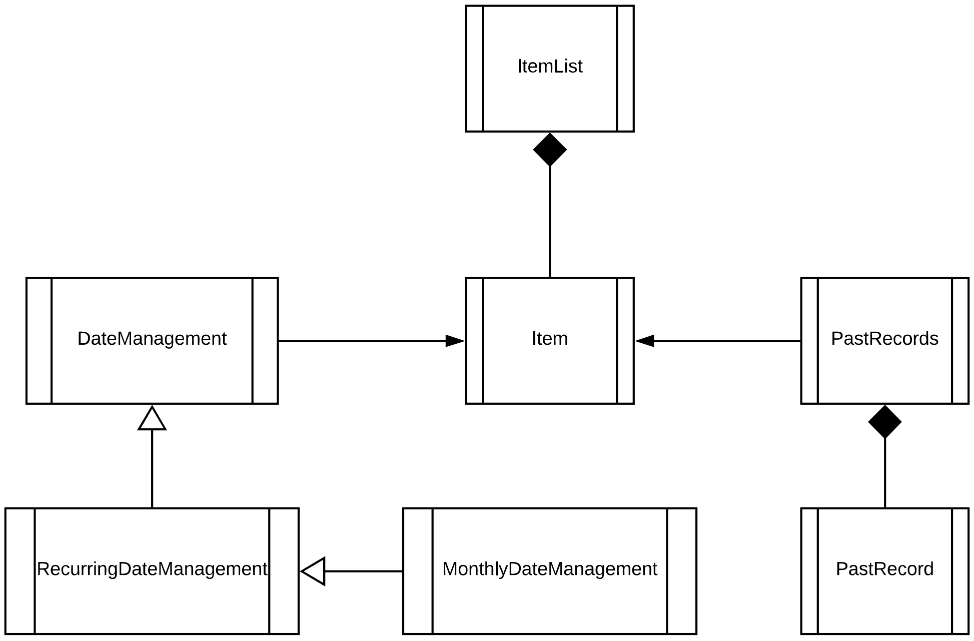


Figure 1: Simplified back-end Data Model

The data model show above is a simple representation that shows the relationships between the TypeScript classes that are used in the server implementation. These classes represent the actual model classes with no regard for the functional modules that are used to perform operations on these classes.

A more detailed outline of each individual class is included below. Accessors and Mutators are omitted for brevity. Most classes include *formatForSaving* and *formatForDelivery* methods. The format for saving method simply serialises the data structure into a JSON structure that can be saved in the database. The format for delivery method is used to perform the calculations necessary to deliver meaningful data to a client.

PastRecord

This class maps to a single entry under the *past* key in the items node. The main function of this class is to provide a method of calculating the difference between the budgeted amount and the actual amount.

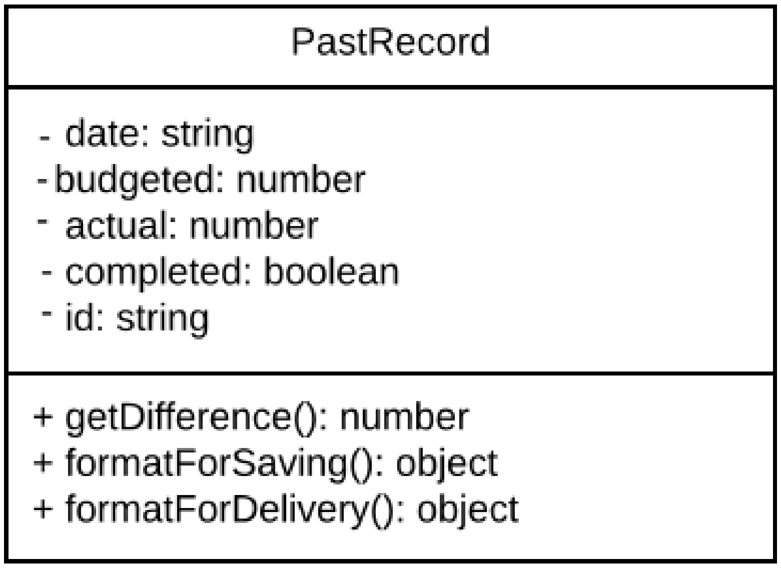
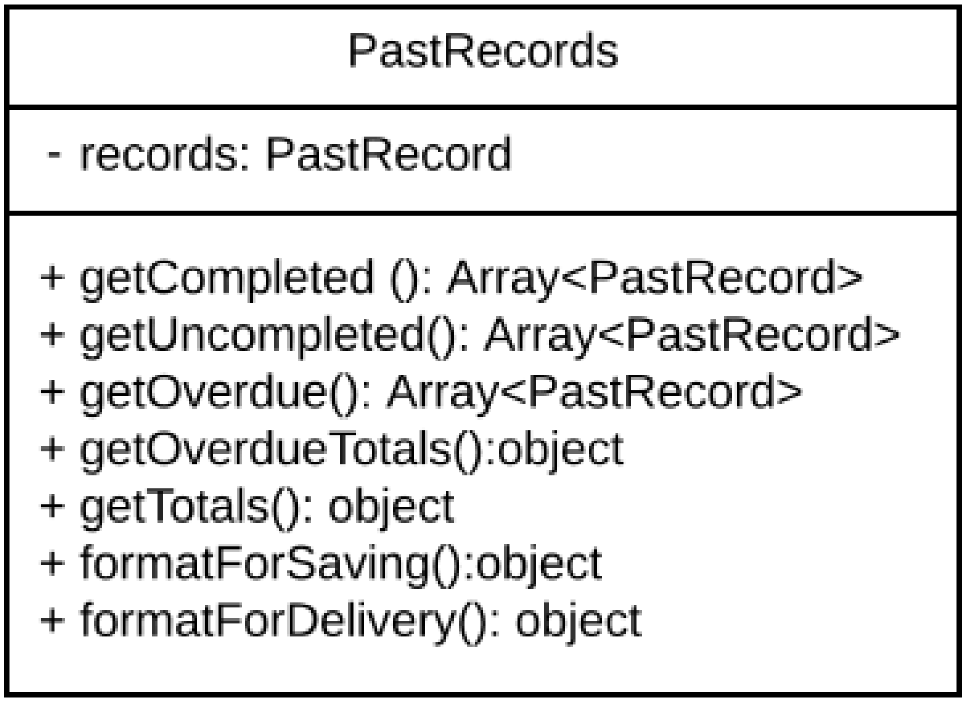


Figure 2: PastRecord model

PastRecords

This class is used for the aggregation of a list of past record instances. The main use of this class is to store all of the past records for a single item, but it could be used for performing aggregation of any list of past records.


PastRecords
- records: PastRecord
+ getCompleted (): Array<PastRecord>
+ getUncompleted(): Array<PastRecord>
+ getOverdue(): Array<PastRecord>
+ getOverdueTotals():object
+ getTotals(): object
+ formatForSaving():object
+ formatForDelivery(): object
Figure 3: PastRecords Model

DateDetails

This is not really a class, but a type. I used this to simplify passing details to the DateManagement constructor. In TypeScript, a type is similar to an interface, except that instead of a class implementing it, it is used to enforce a structure on a plain JavaScript object.

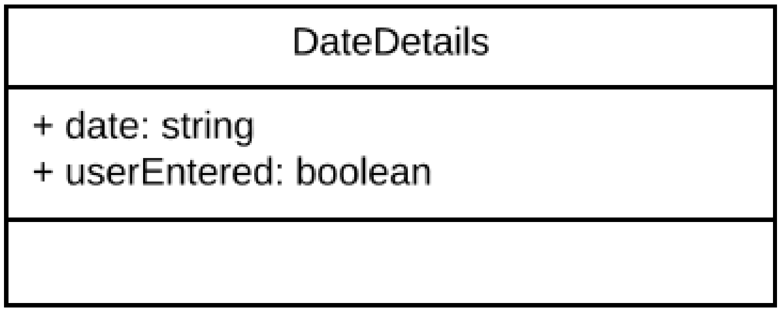
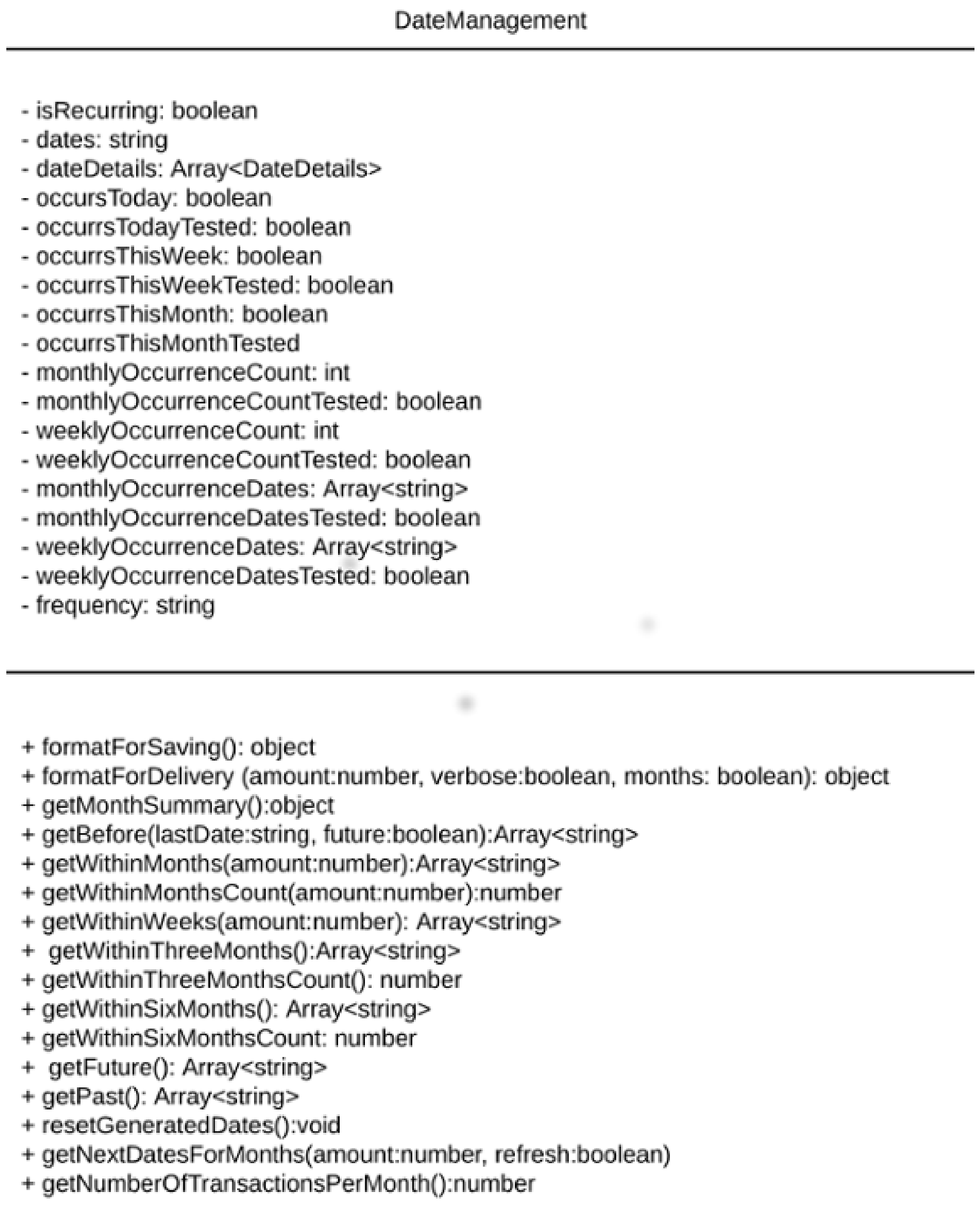


Figure 4: DateDetails Type

DateManagement

This is the largest class in the server. The purpose of this class is to provide a base class that will handle all of the data pertaining to dates that an item will occur. Non-recurring dates will be instances of this class and recurring dates will be an instances of a subclass. Many of the methods of the class may seem superfluous in the context of this class, such as fetching future dates when the class is not concerned with recurring dates, but they are there to provide support to sub-classes.



```
DateManagement

- isRecurring: boolean
- dates: string
- dateDetails: Array<DateDetails>
- occursToday: boolean
- occurrsTodayTested: boolean
- occurrsThisWeek: boolean
- occurrsThisWeekTested: boolean
- occurrsThisMonth: boolean
- occurrsThisMonthTested
- monthlyOccurrenceCount: int
- monthlyOccurrenceCountTested: boolean
- weeklyOccurrenceCount: int
- weeklyOccurrenceCountTested: boolean
- monthlyOccurrenceDates: Array<string>
- monthlyOccurrenceDatesTested: boolean
- weeklyOccurrenceDates: Array<string>
- weeklyOccurrenceDatesTested: boolean
- frequency: string

+ formatForSaving(): object
+ formatForDelivery (amount:number, verbose:boolean, months: boolean): object
+ getMonthSummary():object
+ getBefore(lastDate:string, future:boolean):Array<string>
+ getWithinMonths(amount:number):Array<string>
+ getWithinMonthsCount(amount:number):number
+ getWithinWeeks(amount:number): Array<string>
+  getWithinThreeMonths():Array<string>
+ getWithinThreeMonthsCount(): number
+ getWithinSixMonths(): Array<string>
+ getWithinSixMonthsCount: number
+  getFuture(): Array<string>
+ getPast(): Array<string>
+ resetGeneratedDates():void
+ getNextDatesForMonths(amount:number, refresh:boolean)
+ getNumberOfTransactionsPerMonth():number
```

Figure 5: DateManagement Model

RecurringDateManagement

This class is used to store instances of recurring items. This class directly inherits from DateManagement. This class is mostly used for instances of weekly and daily dates, since both of these implementations have only a single way of incrementing dates. In theory this class should never be used for monthly occurring items, since there is a more specialised subclass for those types of items, but this class does enforce a modicum of sanity in the case of a monthly item being made into an instance of this class. In those cases, the date will simply be incremented by thirty days. This class also overrides some of the methods from its parent class (DateManagement)

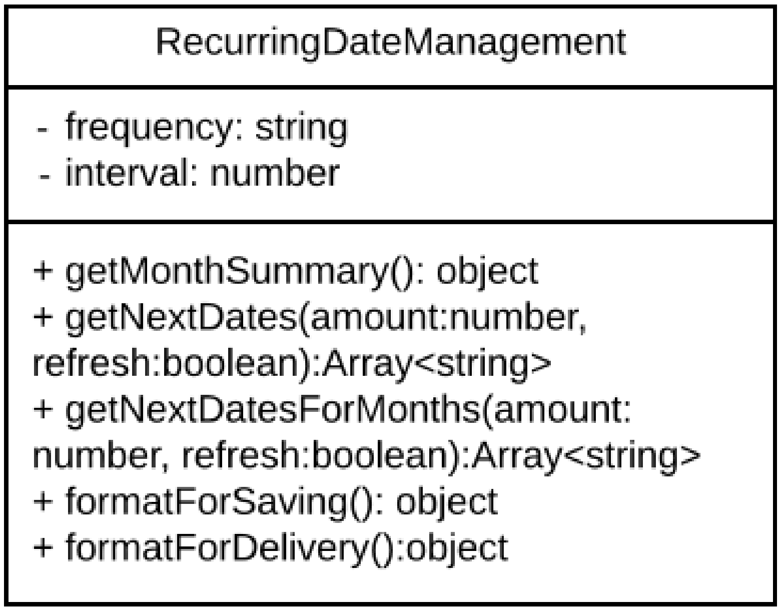

RecurringDateManagement
- frequency: string
- interval: number
+ getMonthSummary(): object
+ getNextDates(amount:number, refresh:boolean):Array<string>
+ getNextDatesForMonths(amount: number, refresh:boolean):Array<string>
+ formatForSaving(): object
+ formatForDelivery():object

Figure 6: RecurringDateManagement Model

MonthlyDateManagement

This class is used for dealing with transactions that occur on a monthly basis. The reason for a separate class for monthly transactions is that monthly transactions have more possible ways in which they could be implemented. A weekly transaction can only occur once every number of weeks, as can a daily transaction, but a monthly transaction could occur on a set date every month, at a specific part of the month (such as beginning, middle or end) or it could occur on a specific day of a specific week in a month (such as the first Tuesday of the month.)

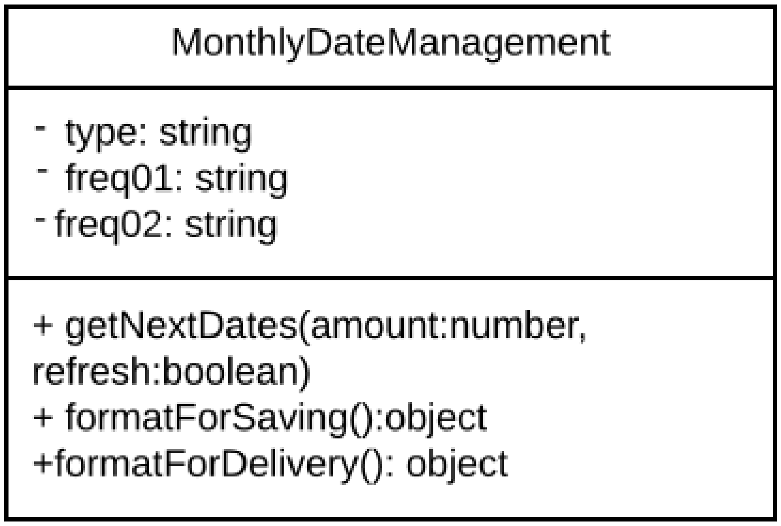

MonthlyDateManagement
- type: string
- freq01: string
- freq02: string
+ getNextDates(amount:number, refresh:boolean)
+ formatForSaving():object
+formatForDelivery(): object

Figure 7: MonthlyDateManagement Model

Item

This class is the one that is mapped to each individual item in the items node in the database. This acts as the aggregation point for all of the aforementioned classes. This class is used to perform aggregations and calculations for each individual item stored by a user. Since the classes used for managing dates have no concept of the financial aspects of the item, this is where the totals for items across classes are managed. This class also stores all of the past records of items and allows for the aggregation calculation of costs of future items.

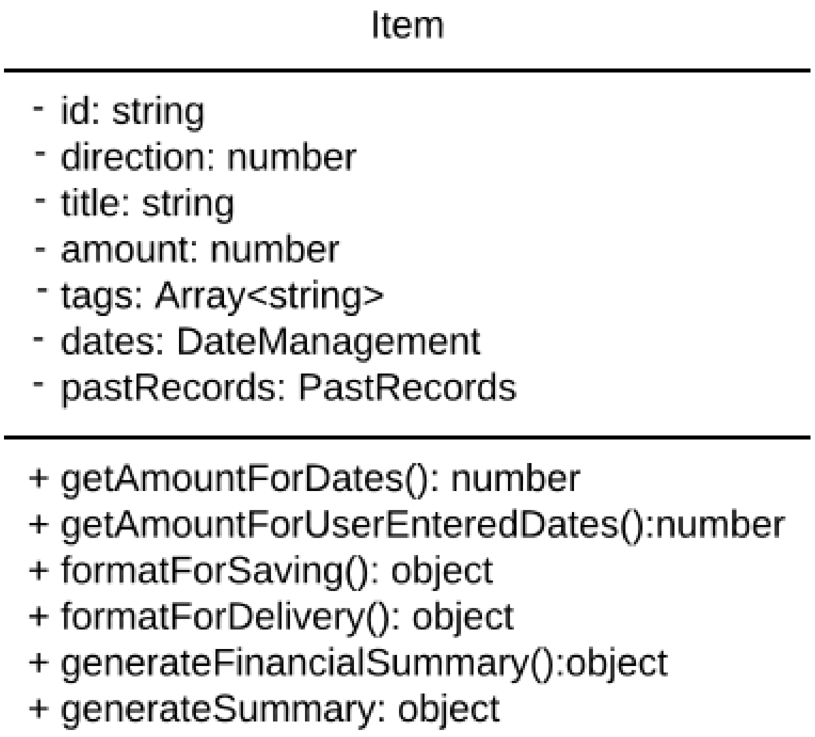

Item
- id: string
- direction: number
- title: string
- amount: number
- tags: Array<string>
- dates: DateManagement
- pastRecords: PastRecords
+ getAmountForDates(): number
+ getAmountForUserEnteredDates():number
+ formatForSaving(): object
+ formatForDelivery(): object
+ generateFinancialSummary():object
+ generateSummary: object

Figure 8: Item Model

ItemList

This class contains a list of items and performs aggregations and calculations across that list. Its main purpose is to serialise a list of items for delivery to a client. To do this it invokes the various aggregation methods found in the other classes to create a data object which can be transmitted to the client. This class usually represents either the entirety or a subset of the a specific users items node.

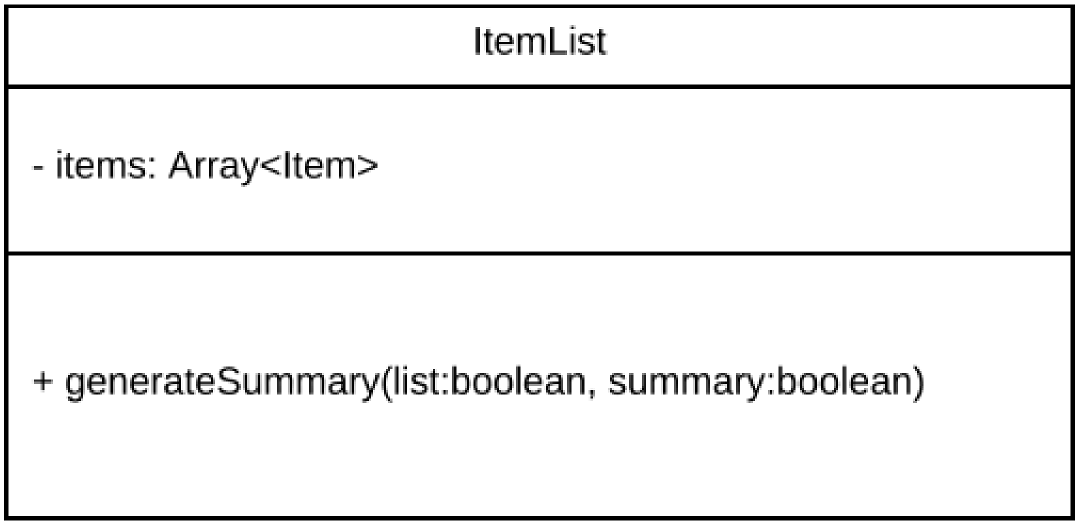


Figure 9: ItemList Model

Old Data Model

Until I was roughly two thirds of the way through the semester, the system used a far more simplified data model, and applied a functional paradigm to try and generate the data that was sent to the client. This turned out to be far less efficient and flexible than the current model. This original data model is preserved under the *transactions* node in the database and in the *finance* app in the server side implementation. However, there is some more client side functionality available that matches this data model.

Client

The client application used to interface with the database is a Progressive Web Application. This means that it utilises service workers under the hood to pre-fetch assets. It also caches HTTP responses so that the data is available even when the user is offline.

This application is also built to be performant. The build process utilises Webpack to bundle modules into chunks that can be loaded one at a time as necessary. In browsers where the service worker spec has been implemented, these chunks will be loaded ahead of time, in the background, when network utilisation is low.

Security

This application is secured using Firebase. Firebase provides an abstraction of common security services. This means that when a user creates an account with an email and password combination, the data is stored in Firebase and a token is returned. This token can then be sent with each request to the back-end and validated by the a Firebase module. This has the added benefit of automatically having the server be aware of the user who made the request.

Firebase also negated the need for me to provide a custom implementation to check if the user was logged in when they re-opened the site. This was automatically done by Firebase on application start. This prevented me from needing to store some kind of cookie or token inside the browser. It also meant that my security was backed by Googles authentication infrastructure, which is quite mature and extremely robust.

All communications are also secured using SSL/TLS. This means that all communications between the browser, presentation server and back-end API are encrypted to prevent sensitive information from being leaked.

Testing

For this project, I followed a Test-Driven Development pattern almost exclusively at first. I wrote all of my unit tests before I wrote the code and I also wrote end-to-end tests for the client that walked through entire operations. As time got shorter, I found that I had less time to implement testing, which led to me only testing complex operations as a quick way to ensure that they worked, and to catch regressions.

On the server, I used Mocha as a test-runner to actually run the tests, Chai as an assertion framework to make assertions about my expectations for the code and Sinon for stubbing/ mocking external events such as database calls. This is a fairly common testing stack for JavaScript applications.

On the client I used the same stack, except that I added Karma to connect Mocha to various browsers, and vue-test-utils to shallowly render Vue components. This allowed me to render my Vue components in a series of headless browsers to ensure that the application was functioning as expected. This is an excellent method for ensuring that UI's work as expected globally, but the problem is that UI testing is, by its nature, very brittle, as I was testing what was appearing on the screen. This meant that whenever I changed the layout of a component, I also had to update my tests. This was one of the biggest factors in reducing the amount of testing that I did.

I used Testcafe for running end-to-end tests. I found this to be an extremely useful tool, but the concept of end-to-end testing is somewhat moot for this type of project. The general idea would be that a client would provide me with an E2E test script and I would have to make it pass. As I was my own client, it seemed somewhat time-consuming to write these test scripts before I implemented my solution.

Where the testing really came in useful was in the back-end and in the parts of the client that integrate with the back-end. When writing these components, I still followed a TDD methodology, whereby I wrote my tests first to determine the shape of the data after a particular operation, and I then wrote the implementation to make the test pass. I feel like this saved me quite a lot of time, as I could write a set of functions designed to perform operations on data before it was added to the view, and test them without having to actually implement the view.

Methodology

In order to build this project, I attempted to use some agile methodologies. This was particularly difficult as it is a solo project over a very short time frame, and the majority of the analysis was done in the first twelve weeks of the year. This made it difficult to accurately assess each sprint as I had so little time between each sprint.

I had originally intended to work with three sprints of four weeks each to cover the fourteen weeks with some additional time, and then use the weekend at the end of each sprint to assess my progress. As it turns out, I underestimated the workload of the final year, and was unable to accurately assess how the individual components of the application were working together, as I had to spend my weekends working on other assignments.

Sprint One – January 10th to February 4th

My first sprint was primarily focused on converting my client side prototype into something more workable. My initial prototype pretty much just covered authentication in the client, as well as my planned styling. It also had a basic about page, home page and empty profile page.

The first step in modifying this prototype was to add caching of my material design fonts and stylesheets. I did this by adding some configuration to my Webpack build that would allow assets requested from URLs to be served from the cache first, and then over the network if those assets did not exist. This meant that the stylesheets would be available even when the application was offline.

I then updated my CSS framework (Veutify) as it had passed its first milestone release (v1.0.0). This was very important, and was something that I was banking on when I chose the framework. The reason this was so important was that, since NPM modules (should) adhere to the semver specification for semantic version numbering, any release lower than 1.0.0 should be considered unstable. This meant any upgrades of the library could introduce breaking changes. Once a library passes it’s 1.0.0 release, only updates to the major release number should introduce changes. This meant that reinstalling dependencies could break things in the application since NPM, by default, will install the latest version with the same major version number. For example, if the application was using Vuetify v0.15.4 and I reinstalled the dependencies, it could upgrade to v0.16.0, which in a release before 1.0.0 could introduce breaking changes.

The next step that I made was to create a group details component that would be integrated into the profile page. This component was designed to show details about the group that the user belonged to. There was also a form created that could be used to create a group. Ultimately, this feature was abandoned, as I realised that it would be quite difficult to implement, because of the way that Firestore organises its data. Firestore is designed in such a way that it makes it very easy to secure data so that only the user it belongs to can view it, but this makes it quite difficult to allow groups of users to access the same data.

At this point I realised that some of my code for handling forms that were contained in dialogs was getting quite repetitive. There was quite a bit of code that was required for any form that was contained within a dialog. This included actions such as closing the dialog, resetting the form and sending events about the form data. To counteract this, I created a Vue mixin which encapsulated this logic. This mixin could be added as a property of any Vue component, which meant that all of the logic in the mixin would be added to every component.

Once this was done, I realised that a lot of the logic of my forms was quite repetitive too. These actions included resetting the form values, resetting the validator and emitting events when the form values changed. I decided to do what I had done before and create a mixin to handle this repetitive logic.

The next step was to create a new UI so that a user could set a device as trusted. The reason that I needed this was that in order for Firestore to be available offline, I had to allow it at the application start. According to best practice, it is best to ask the user if they are using a trusted device, to prevent data being saved on a device they do not own. This feature was also ultimately abandoned, as I decided to simply integrate it into the login dialog.

Once I had a UI in place for asking if a device was trusted, the next step was to store this preference on the device. To do this, I used Local Forage. This allowed me to transparently use the best storage media on the current device. For example, at time of writing, this would be IndexedDB then WebSQL then localStorage. This meant that if a user was using a browser with IndexedDB included, this preference would be stored here, if not then it would fall back to WebSQL and if that was not available, the preference would be stored as text in localStorage.

Once this was completed, I needed to make this data available throughout the application. To do this I created a new Vuex store module to manage preferences. This meant that I could read this at application start and have it available in memory to any component that requires it.

The next thing I did was to update my dialog mixin so that it could cache the state. The reason for this was to allow multi-step forms where the user could click the back button and have the previous form be already populated with the data they entered. This was tricky refactor, as I now had to check when the form was instantiated if there was cached data available to populate the form. I was doing this as I had intended to create a multi-step sign up form which would allow the user to create a new group when they initially signed up. As previously mentioned, the group feature was abandoned, making this refactor redundant.

The next step was to refactor how the updates to localStorage were handled. Initially, there was a function call which wrote to localStorage and then dispatched an update to the store. This broke the application at first, as the module that contained the function was trying to import the store before it was initialised. This meant that to get it working, I had to write to the device, and then dispatch an event to read the value that had just been written. This had obvious performance implications. With this refactor, I instead moved the function call that wrote to the device into the store. This meant that I dispatched a single value to the store, which was written to both the device and the store at the same time.

The next step was an application architecture refactor. I did this to properly separate application components based on their function. Originally I had it grouped so that each “feature” was in its own directory. This meant that all of the store modules, UI components and services were grouped by feature. This lead to issues where I was importing the services and and store modules from one feature to another. Instead I decided to re-organise so that I had a data directory for fetching data, a services directory for functional services, an events directory for handling events that would be dispatched throughout the application and a UI components directory for holding all of the UI components. I maintained the initial idea of organising by feature underneath these directories.

The next step was to add Flow to the application for static typing. This allowed me to specify types of function parameters. I chose Flow over TypeScript as it can be integrated into existing JS files much more easily and required less refactoring.

The next step was to begin work on the form that would be used to add financial details to the application. For this I created a generic form component that had all of the details required to add financial details to the system. This form could be used for both incomes and expenses as they both used identical data structures. I then created two wrapper components that contained this form. These wrapper components were the ones that gave context to the form. Since the data structures were identical, these components could dispatch an event to save the data from the form as-is in the correct location.

I also created my first iteration of the back-end during this sprint. This was ultimately a failure, mainly due to my inexperience with how Cloud functions are structured. I created this back-end and used Webpack to bundle each “app” for deployment. Overall, I was never truly happy with how it was progressing and with my architecture in general. What finally finished it was that I decided to enforce structure operations that had a high probability of failure. I did this by creating a Result class which could hold a boolean to say if an operation was a success or failure along with the data/error generated by the operation. When I tried to refactor the server to use this new class, I realised that it would be difficult to ensure that this happened everywhere in a loosely typed language. To overcome this, I tried to integrate Flow to type check. It was only when I got this up and running that I realised how deep this refactor would go. I would have had to refactor almost every single function (~50 functions) in the application. I decided to cut my losses and start again.

Sprint Two – February 4th to March 4th

Back-End

This sprint began with me re-starting my back-end. This time I opted for using TypeScript from the get-go. Firebase Cloud Functions has an option to generate the base for the project using TypeScript. I availed of this feature and was able to generate a default configuration that was set up to transpile TypeScript to JavaScript.

The very first step in creating this back-end was configuration. The first thing that I did in terms of configuration was to set up unit testing. I felt that this was very important for the back-end, as so much of it was data driven. I needed to have the testing infrastructure in place to ensure that I could test functions as I wrote them, instead of needing to build a whole function chain from the endpoint to the database to test manually. Once that was completed, I also had to configure the Firebase-Admin API to be connected to the correct project.

The first coding step was to create the Result class which was the class that would be used to determine if an operation was successful, and return data or an error depending on the outcome of that operation. I did this first as this was the issue that led me to abandon my previous attempt at the back-end.

The next step was to handle authentication. Luckily Firebase provide a generic middleware function on their example page that can be used before each request. This function checks if an Authorization header is present on the request, that it conforms to OAuth 2.0 specification and that it is a valid Firebase token. If any of these are false, this function returns the string “Unauthorized” with a status of

403 to the client. If all of the checks pass, the ID of the user who issued the request is appended to the request object and routing continues.

After this, I created a the Transaction model class. This was the base class for all transactions. This was a pretty simple class, and was really only used to enforce structure on the objects that would be operated on. It did have a clone method to return a new copy of itself, to prevent mutations to the initial object, in line with the functional paradigm.

The next step was to create a service module that would be used to perform operations on the transaction model. This was quite possibly the most difficult part of the entire project. I built a module which contained mostly “pure” functions that were designed to operate on instances of the Transaction class. These operations were mostly based around the concept of generating future dates based on the date and frequency in each individual transaction. It was here when I first began to question my initial data model. This was because I had it set to save the next due date of an item when the object was created, then when future dates were requested, I needed to increment this date to get to the nearest future one, and then start generating dates. This took a lot of figuring out as, for example, if someone had entered a daily transaction six months ago, I didn’t want to generate six months worth of old dates before beginning to generate new dates. Instead I tried to estimate how many transactions would be needed, immediately increment by that amount, and work backwards or forwards to the current date before I started generating transactions that were required.

Front-End

While I was working on the back-end, I was also trying to work on the front-end. The first step was to actually write the transactions entered in the form to the database. I decided to forgo sending the data to the back-end to write to the database, instead I wrote directly from the client. The ability to do this was one of my major drivers for choosing Firebase. Ultimately, I believe this to have been a mistake, as it added a lot of complexity to the client, when in reality, the client was never the important part of the application.

Once I had the data being written to the database, I started to work on how to display it. This turned out to be very complex, as I wanted to have the ability for a user to generate a list of transactions for a set number of months. In order to do this, I simply fetched the full list of entered transactions when this view was loaded, and then added a button to fetch another six months worth of records, which were appended to the dataset.

I also wanted to have the ability to do some advanced filtering and sorting. Luckily, filtering by text and sorting was particularly easy to accomplish thanks to Vuetify. All of their datatables are automatically sortable by each column, and they come with a *search* parameter which I could bind to a text field that automatically filtered the table based on the contents of the search field.

It was more complex to add more advanced filtering. I wanted users to have the ability to filter by date or by type. To do this I created a side navigation component that would only appear when the user was viewing data. This contained the controls for filtering the dataset based on type, and had the ability to bring up a calendar that would allow the user to filter by date or by month. I filtered the initial dataset using these controls, which meant that the free text search on the table could be used to further filter this data.

I also created a summary component that could be used to show summaries of all of the data that had been fetched so far. Once I had this completed, I added the ability to show or hide it based on the contents of the side panel.

The next step was making the site more responsive on mobile. My main problem was that when the side nav was opened, it pushed everything else off the screen on mobile. On desktop, the contents of the screen shrank slightly when this navigation drawer was opened, but there simply was not space on mobile. To counteract this, I added a property to the drawer that it would float out over the page content on mobile.

The next step for making the site more responsive was to replace the data-table with a data-iterator. This is a new component from Vuetify, which work similarly to a data-table, except that it displays the data on a series of cards instead of in a table. I made this the default view on small screens, but kept the option for users to switch between views.

Sprint Three – March 4th to End

Early into this sprint, I realised that much of my earlier work would have to be discarded. I realised this because I tried to update my back-end data model to include details about past transactions and I ran into quite a lot of difficulty. To counteract this, I immediately began work on the more complex data model outlined in the above sections. It turned out to be much easier to manage my data model using an Object Oriented approach instead of a functional one. I started at the top of the data model, creating the Item class first. I then worked down through all of the date management classes to add the functionality needed to generate dates. After that, I created the past records class to handle older records. Because I was running so short on time, I didn’t have the luxury of designing my schema ahead of time. The final schema grew in an ad-hoc manner as more data was added to the model. Once I had my schema created I started creating API endpoints that could be used by clients to request data. These were very simple functions to write as they simply deal with extracting the necessary parameters from the request in preparation for fetching the correct data. Once I had the routes in place, I created a service layer that acted as a go-between for the routes and the models. This layer fetched the data from the database and created instances of the models so that the data could be aggregated. The next step in the project was to create a new form that could be used to gather the information to send to the client. This was quite a complex form, to try and deal with all of the possible frequencies that records could reoccur. Despite the complexity, I managed to complete this form with relative ease. The final step in creating this project was to create a simple view to display the data that was returned from one of the endpoints. This was a very simple view that just showed a summary of the upcoming month, and all transactions that were due to occur in this month. I then removed the links for the old view from the navigation.

What I Learned

I learned a lot while doing this project. One of the most valuable lessons I learned was the value of well tested, mature technologies. The biggest example of this was my choice to use a NoSQL database. I feel that, in hindsight, my architecture would have lent itself better to a more structured database implementation. However, I did enjoy the challenge involved in trying to make my chosen implementation work. I also feel that if I were to develop this project again, having built it in a NoSQL technology would give me a new perspective when designing it in SQL.

I also learned a lot about client side performance operations. On reflection, I may have spent too much time working on this aspect of the client, considering it is only a small prototype to showcase the capability of the back-end. However, I learned a lot of small ways to improve the performance of a client side application from compile-time tree-shaking, to asynchronous component definitions, to deferred module loading, to module bundling. This is all knowledge that will come in extremely useful for future client-side development.

I also feel that I learned a lot about testing a software application. Before starting work on this project, I had very little understanding about the process of integrating unit testing into my development work flow. After working on this project I feel like I now have a much clearer understanding of the importance of properly integrating unit tests into an application. I also now have the knowledge implement such a solution in future projects.

One of the most profound lessons that I can take away from this project is the importance of carefully reviewing the software design paradigm to be used. For dealing with transactions initially, I was using a functional paradigm. It the time I was happy with it, but it was only when I redesigned my data-structure that I migrated to using a mix of Object-Orientation and Functional paradigms. This was when I realised how much easier it was to handle dates inside the local state of an class instance, rather that simply performing functional operations that returned a plain JavaScript object containing the results.

Conclusion

In conclusion, I really enjoyed working on this project. It really opened my eyes to how a software project grows in complexity as it grows in size. I always knew that this would be the case, but I hadn't realised how much my progress would slow as the application grew.

I feel that my decision to use Cloud Firestore as a database was a mistake. The main reason for this is that I now feel that the data structure is more suited to SQL. I feel that I would have had a much easier time in managing my data in this way. I had looked at the possibility of implementing an SQL database when I redesigned the data model, but because I was using the free tier of Firebase Cloud Functions, I was only able to access Google API's on my server. I looked into using Googles Cloud SQL service, but this does not have a free tier. So my options were to pay for Firebase, pay for Google Cloud SQL or redesign my entire system architecture. I opted to stick with Cloud Firestore.

Even if I had decided to stick with a NoSQL database, I feel like Cloud Firestore was a poor choice. It is currently still in beta, and its query implementation is quite poor. This made it hard to properly query collections, and meant that I had to load entire collections into memory and filter them there. I had to work quite hard to try and reduce my memory footprint, to prevent my server from needing to be scaled unnecessarily.

Firestore also does not support deep fetching, which is something that is not mentioned, at all, in the documentation. This means that when I query, for example, the *items* collection the nested collection *past* is returned as an empty array. First of all this caused me quite a bit of confusion as I thought my implementation was broken. Then once I realised my problem, I realised the full implications of this. In order for me to fetch all past records for a user I would have to query the database for all items, iterate through those items and perform another query for each to fetch the past records. To put this in context, if a user had 100 separate item records in the database (not too out of line, considering the amount of different types of food a household might consume) fetching all past transactions would require 101 separate queries to the database when one SQL or MongoDB query would suffice. In a production application, I could not justify this overhead.

Aside from that, I was very happy with my choice of infrastructure. Using Google Cloud Functions was probably unnecessary for this project, as it will not be receiving the traffic to justify the need for such a scalable server instead of a standard REST API. However, it was a very easy service to use, perhaps even easier than hosting a standard REST API. It also provided total SSL for all communications, unlike Heroku, which only provides SSL as far as their own load balancer.

I was also happy with my choice of client side framework. Vue is very easy to use and I felt like it had a much lower learning curve than the other options that were available. However, I also feel that I spent too much time working on performance optimisations for the client side. This was because of the fact that the client is only for demonstration purposes, the real power of this application lies in its back-end infrastructure. I feel that perhaps I should have stuck with the default configuration provided by Vue CLI, rather than tweaking it to tease out the maximum amount of performance.