Home Resource Planner Application

Home Resource Planner (HRP)

Status Report (Semester One)

Joe Wemyss

20068336

Supervisors Name: Catherine Fitzpatrick

BSc (Hons) Software Systems Development

Home Resource Planner

DESIGN SPECIFICATION

Joe Wemyss | Software Systems Development | December 2017

Contents

Figures

Introduction

This is the initial design specification for my application, Home Resource Planner. This document outlines the steps taken in the initial planning stage of creating this application. First the socioeconomic drivers for this application must be examined. This can then be used to provide a set of goals that the application will attempt to adhere to throughout the development lifecycle.

Once the goals have been decided upon, it will be necessary to perform some analysis on potential users for the application to better understand the applications target audience. Then an assessment of development methodologies can be carried out.

Once the goals, users and methodologies are clear, it will be possible to turn the goals into a set of possible features to be implemented. These features will then be converted into a set of functional requirements. It will also be necessary to define a set on non-functional requirements to ensure that the application performs as expected.

Once both the functional and non-functional requirements have been defined, it will be possible to examine potential technologies to decide what is the best software stack for the proposed application. This leads on to designing a reference infrastructure that will form the underlying system infrastructure for the application.

Once the infrastructure has been defined, it will be possible to define an underlying data model for the back-end of the system. Once the data model is clearly understood, it will be possible to determine how best to proceed with delivering this data to the user, vis-à-vis a client-side application. In order to define how this data should be delivered, it will be necessary to compare and contrast the possible technologies that could be used.

Project Goal

MOTIVATION

The motivation for this project is to provide a unified data management service for the home. The value of data in the modern age cannot be overstated. The amount of data generated is growing exponentially (it is estimated that by 2020, we will be generating 1.7MB per person per second, globally (Kumar, 2017)The amount money being spent by companies to manage this data is also growing ((Gartner, 2017), says that IT spend in business will grow by 2.4% in 2017, to $3.5 trillion)

A reasonable assumption from these facts would be that as technology becomes more prevalent in our every-day lives, more data will be generated. Another assumption that could be made is that since business spend on managing data is growing exponentially, businesses find some value in successful management of this data. This leads me to my third assumption, that ordinary households can also gain value from the data that they generate, if it is managed properly.

GOAL

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

For the system to be considered a success, it must be easily usable by everyone. The UI must be simple enough that anyone can grasp, as well as accessible for people with disabilities. The underlying data structures must be flexible enough to deal with changing user requirements, as well as adaptable household processes. Application flow must also be flexible to help with user experience. An application that feels rigid is never conducive to good UX (Nagy, 2015).

The overarching goal of this project is to provide an easy to use 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 user must always be the main driver for changes in the application. If a sample user group is not regularly interacted with, the project runs the risk of not being tailored for its target market.

The overarching goal of this project is to provide an easy to use 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.

User Analysis

TARGET AUDIENCE

The target audience for this application will be anyone who has a need for extra transparency in how their household is run. The basic idea is that the users would be part of a group, which represents a household. Certain resources would be allocated for individuals, and some on an entire household basis. There will be varying degrees of transparency into other transactions, based on how each user account is configured.

Some members of the household will be designated as “admins” and will have access to the most data about other members of the household. Members who are not admins will have to be able to mark transactions as “private” so that only summary data of private transactions are available to the admins.

This structure will allow the application to be suitable for the largest possible target audience. Therefore, ease of use will be so important, as the system may need to be used by younger members of the household, as well as older members who may not be entirely technologically literate.

Methodologies

With the way the year is broken down, Semester One for analysis and Semester Two for development, the most obvious methodology for building this project would be the Waterfall Method. I am reluctant to use this method however, since it has a reputation for being very inflexible (ExpertsExchange, 2014). Since this project will have constantly changing requirements, a more iterative approach is needed.

The most common implementations of Agile, such as SCRUM (ScrumGuides, 2016) and Kanban (LEANKit, 2017), are also not possible for this project, since I will be working solo on it. Most Agile implementation is very team oriented, thus making it unsuitable for solo developers.

Since neither Waterfall nor true Agile is possible, I decided to go with a hybrid approach. I will use the first 12-week period as requirements gathering and definition, while working on building a prototype. I plan on following Agile principals while developing this prototype, such as Test-Driven Development(TDD)/ Behaviour Driven Development(BDD), LEAN (Ambler, 2016)CI/CD.

The prototype will hopefully be able to serve as a LEAN Minimum Viable Product(MVP) (TheLeanStartup, 2017) that can be used for beta testing. This prototype will also act as a Technical Feasibility Check for the project, where I will test out the technical specifications of the languages and frameworks that I plan to use.

The second half of the year will be given over to development. Since it will only be a 12-week period to get the final product built and deployed, sprint cycles will have to be kept short and fast. In conjunction with LEAN principals, paperwork that does not directly add value to the project is considered waste, and must be eliminated as much as possible. This means that reports and documentation that accompany this project must be concise and succinct to minimise time wastage.

As an actual development methodology, I plan to build the application from the outside in. I will build screens first, stubbed with dummy data if they are data driven screens. Once I have an idea of how I want my data to be presented, I will build corresponding API routes on the backend. When building the API routes, I plan to build my DB models first, based off the stubbed data in the client app, then build the controller that will handle the route before building the interceding service layer to match up the two. This is a common pattern for API design.

I will be aiming to follow TDD/BDD methodologies as much as possible throughout the project. This will add to the project overhead, but may reduce the chance of bugs being introduced to the system later in the project.

I also hope to get a Continuous Integration (CI) solution in place for the prototype. This is something that I have never done before, which is why I would like to have it implemented before development begins in earnest. This would greatly reduce DevOps overhead, since the integration server would handle all unit testing/deployment.

Requirements Analysis

FUNCTIONAL AREAS

One of the first tasks of any software project is to define a set of functional requirements. Functional requirements define the functionality, or behaviour, of a system. (Weigers, 2003). The goal of the project can be summed up as:

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

For the project to be a success, the functional requirements must attempt to adhere to this goal as much as possible. To facilitate the development of this set of requirements, I decided to separate the application into several functional areas. Each functional area defines a set of features that are more closely interrelated than most. Of course, these functional areas are purely conceptual for requirements engineering purposes, and will have little impact on physical structure.

Login

The user should be able to signup/login to the application using various social media forms. This feature is included as a functional requirement, as it is a function that the user should be able to carry out, but it also helps to satisfy some non-functional requirements, such as adhering to web standards, security and UX.

A big consideration here is, how will users get into the application initially? Since the application is going to be used for managing a household, how will the initial login flow look for each user in the group? These questions will be critical to determining an authentication strategy.

Track Suppliers

The user should be able to track the suppliers that they make purchases from. They should have some method of being geospatially aware of providers of services in their region. Since the application aims to provide location based services, there needs to be a distinction between whether the suppliers are a location that the user may visit physically, an online retailer or the provider of a service.

The user should also be able to keep a record of their finances spent at each supplier. This function would be dependent on the applications ability to track finance, another functional area.

The user should also be able to keep track of the purchases from these suppliers. This will again be another feature that will need to integrate with the financial planning section of the system, as each purchase of an item will correspond to a financial record. A later stage feature that could be implemented here would be the ability to calculate average cost difference between suppliers, to better enable purchasing decision making.

Track Finance

Transaction Management

This is possibly the most critical aspect of the application, as it feeds into almost every other module in the application. Realistically, every other module exists as a specialised implementation of the finance module. For this reason, the requirements of this module are of importance.

The user should be able to easily add transactions into the system that can be aggregated and presented to the user in various contexts. For example, a user should be able to see their average spend by various metrics such as supplier, physical shop, location and item. They should also be able to forecast future spends based off current spending trends.

There is a critically important non-functional requirement that needs to be considered here, Usability. It is highly impractical to expect users to enter every single detail of every transaction in real time. The data must be dynamic enough to piece together partial and incomplete data that the user can enter easily. This fragmented data must still be able to provide value to the user, else it is worthless. An example of this would be when a user adds a food transaction, they should be able to add a total and then add the items at their leisure.

Budgeting

The user should also be able to create budgets. The ability to create budgets should be able to make inferences about the budget from the user's transaction history. For example, the budget could default to 80% of the average weekly food spend for optimistic forecasts, and 120% of the average weekly food spend for pessimistic forecasts.

The budget should also be able to consider the utilities that the house could face. These utilities could be fixed or variable, so there should be functionality to set optimistic and pessimistic bands for the budget.

Inventory Management

The user should be able to use the application to manage the inventory of the house. This feature set will have tight integration with both the finance and supplier modules. This is because users will use finance to purchase items for the inventory from suppliers.

The inventory management section will be used to track the details of household items. This will mainly be used to track food and clothing in the beginning. It may be necessary to update the types of data stored as requirements change.

The main functionality of the Inventory Management module initially would be to track current supplies of food in the house, to give the user alerts about data like expiry dates and low supplies.

The ability to track expiry dates will be reasonably easy as this will mostly entail somehow manually capturing expiry dates where possible, and setting sensible defaults where not. This again should not be intrusive for the user. It should be a feature that should be available when needed, but hidden when not.

The ability to track the amount of food in the house would be slightly more difficult to implement, as it is difficult to capture data about these items. For example, how could you estimate the amount of salt used daily? To achieve this, historical data will need to be available about average item turnover, to infer approximate reorder times for budgeting purposes.

FUNCTIONAL REQUIREMENTS

From the feature set described above, it is possible to create an initial list of requirements for each feature set for consideration:

Transactions

Transactions	
	CRUD Functionality
	Made up of Items
	Has aggregate totals
	Provide mainly numerical data
	Child relationship with Supplier

Figure 1 -- Transaction Functional Requirements

Items

Items	
	CRUD Functionality
	Represents a single item
	Part of a Transaction
	Part of the current Inventory
	Bound to Separate Supplier to Transaction

Figure 2 -- Items Functional Requirements

Suppliers

Suppliers	
	CRUD Functionality
	May be physical or virtual
	If physical, should have geospatial coordinates
	May also be service providers
	May also be Item manufacturers
	Provide mainly categorical data
	Provide a source to generate Transactions

Figure 3 -- Suppliers Functional Requirements

Inventory

Inventory	
	CRUD Functionality
	Should track turnover
	Should provide usage levels for budget
	Fast changing data
	Needs to be dynamic

Figure 4 -- Inventory Functional Requirements

Budget

Budget	
	CRUD Functionality
	Provides forward outlook based off user input and other features
	Should provide bands for both optimistic and pessimistic outlooks
	Should be able to provide more accurate default forecasts as more data is available
	Provides list of predicted Transactions from most economical Supplier

Figure 5 -- Budget Functional Requirements

Dashboard

Dashboard	
	Provides correlations/aggregations between functional areas
	Should be customisable
	Customisability state should be stored in URL for bookmarking
	Graphical Representation of data

Figure 6 -- Dashboard Functional Requirements

Authentication

Authentication	
	Streamlined
	Integration with 3rd party SSO providers
	Manages both users and groups

Figure 7 -- Authentication Functional Requirements

NON-FUNCTIONAL REQUIREMENTS

Authentication	
	Secure
	Useable
	Third Party SSO Integration
Performance	
	Usable offline
	Lightweight
	Fast Installation
	Scalable
	Responsive
UX	
	Clean interfaces
	Fluid application flow
	Adaptable
	Expected design patterns
	Standardised layout
	Same experience cross-platform
Usability	
	Cross Platform
	Accessibility
	i18n Standardisation

Figure 8 -- Non-Functional Requirements

Architecture

TECHNOLOGY OVERVIEW

Before choosing an architecture, it was of vital importance to perform a technology review of services that could be used to build the architecture. The first technology provider to be reviewed was Firebase.

Firebase

Firebase is a mobile first provider of Infrastructure as a Service (IaaS) from Google. It provides a more light-weight experience than the other providers, but since it is from Google, it provides easy integration with Google Cloud Platform (GCP) (Firebase (A), 2017)

Firebase	
	Authentication
Reference	(Firebase (B), 2017)
Description	A robust Authentication solution that provides a unified approach to dealing with 3rd party SSO.
	Real-Time Database
Reference	(Firebase (E), 2017)
Description	A Key/Value JSON Store that can hold data accessible through Web Sockets
	Cloud Firestore
Reference	(Firebase (F), 2017)
Description	A NoSQL, Collection based database that can be accessed directly from the client
	Cloud Functions
Reference	(Firebase (G), 2017)
Description	Small NodeJS scripts that can be accessed through REST calls, or triggered by events in Firebase
	Cloud Messaging
Reference	(Firebase (I), 2017)
Description	A solution to provide push notifications to applications that support them (Native Apps & PWA's)
	Cloud Storage
Reference	(Firebase (J), 2017)
Description	File storage buckets that can be accessed from the Client
	Hosting
Reference	(Firebase (K), 2017)
Description	Hosting for both static Web Apps and Cloud Functions, with HTTPS

Figure 9 -- Firebase Technology Summary

Google Cloud Platform

Google Cloud Platform is the more mature IaaS solution from Google. It offers a more complete feature set than that of Firebase, but also integrates very well with the services provided by Firebase (Google Cloud (A), 2017).

Google Cloud Platform	
	Cloud SQL
Reference	(Google Cloud (G), 2017)
Description	A scalable SQL solution for storing data on the backend
	Maps Geocoding
Reference	(Google Cloud (E), 2017)
Description	An API that can be used to convert mnemonic addresses into geospatial coordinates and back
	Analytics
Reference	(Google Cloud (F), 2017)
Description	Cross platform analytics
	Big Query
Reference	(Google Cloud (B), 2017)
Description	Scalable, column-store data warehousing
	Maps UI
Reference	(Google (H), 2017)
Description	An API for embedding Google Maps widgets into Client Apps
	Maps Places
Reference	(Google Cloud (I), 2017)
Description	File storage buckets that can be accessed from the Client
	App Engine
Reference	(Google Cloud (C), 2017)
Description	Scalable IaaS Application Server

Figure 10 -- Google Cloud Platform Technological Summary

AWS

Amazon Web Services (AWS) is probably the most mature and well-known provider of IaaS. They provide a very comprehensive list of technologies that can be used to build web applications. (AWS (A), 2017)

Amazon Web Services	
	EC2
Reference	(AWS (B), 2017)
Description	Application Server as a Service
	Lambda
Reference	(AWS (C), 2017)
Description	Severless functions triggered by HTTP or AWS
	Lex
Reference	(AWS (D), 2017)
Description	AI Platform that forms the basis for Amazon Alexa
	API Gateway
Reference	(AWS (E), 2017)
Description	Provision and secure endpoints for EC2/Lambda systems
	Aurora
Reference	(AWS (F), 2017)
Description	SQL based DBaaS
	Dynamo
Reference	(AWS (G), 2017)
Description	NoSQL DBaaS
	S3
Reference	(AWS (H), 2017)
Description	Scalable File storage buckets

Figure 11 -- Amazon Web Services Technology Summary

Other

Other	
	Heroku
Reference	(Heroku, 2017)
Description	Application Server as a Service Provider
	Surge.sh
Reference	(Surge, 2017)
Description	Client-Side Website Hosting
	Algolia
Reference	(Algolia, 2017)
Description	Free Text Search across data models
	Now.sh
Reference	(Zeit, 2017)
Description	Application Server as a Service Provider
	Atlas
Reference	(MongoDB Atlas, 2017)
Description	MongoDB DB hosting
	mLab
Reference	(mLab, 2017)
Description	MongoDB DB hosting

Figure 12 -- Miscellaneous Technology Summary

Chosen Solution

In terms of infrastructure platforms, it was decided to begin with Firebase and integrate GCP as the application grows. I made this decision as Firebase seemed to have a lower entry barrier than the other two providers mentioned, as well as a very easy to use client-side API. Since both Firebase and GCP are provided by Google, they are designed to be integrated with one another. This was my primary reason for not using AWS as my infrastructure provider.

INFRASTRUCTURE

The most likely choice of infrastructure is outlined below. The way I designed this was to start with the most complete target architecture and scale back to essentials.

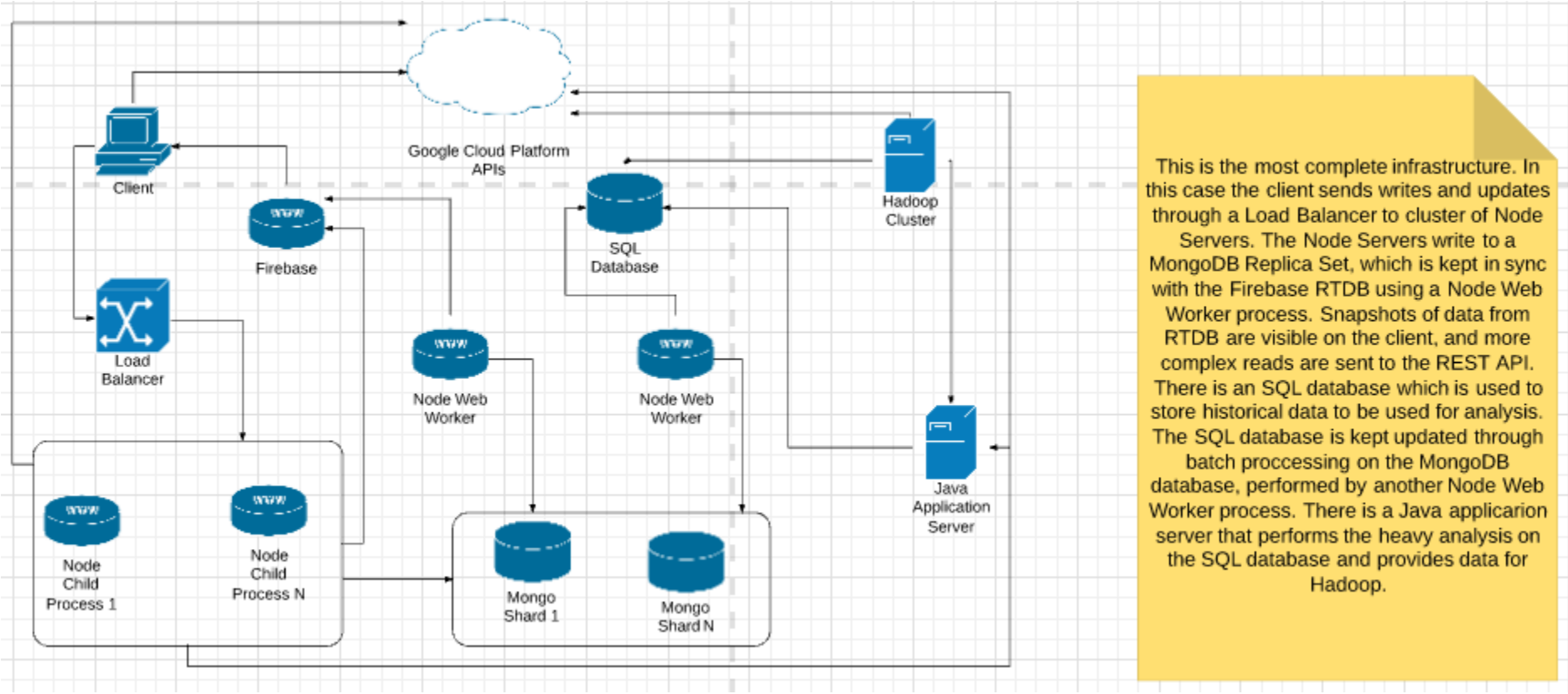


Figure 13 -- Infrastructure Diagram One

The above image is the most complete system landscape. Below is the minimum required landscape to service the functional requirements. Intermittent steps define a clear migration path between the diagram above and the diagram below, and can be viewed in the appendix of this document

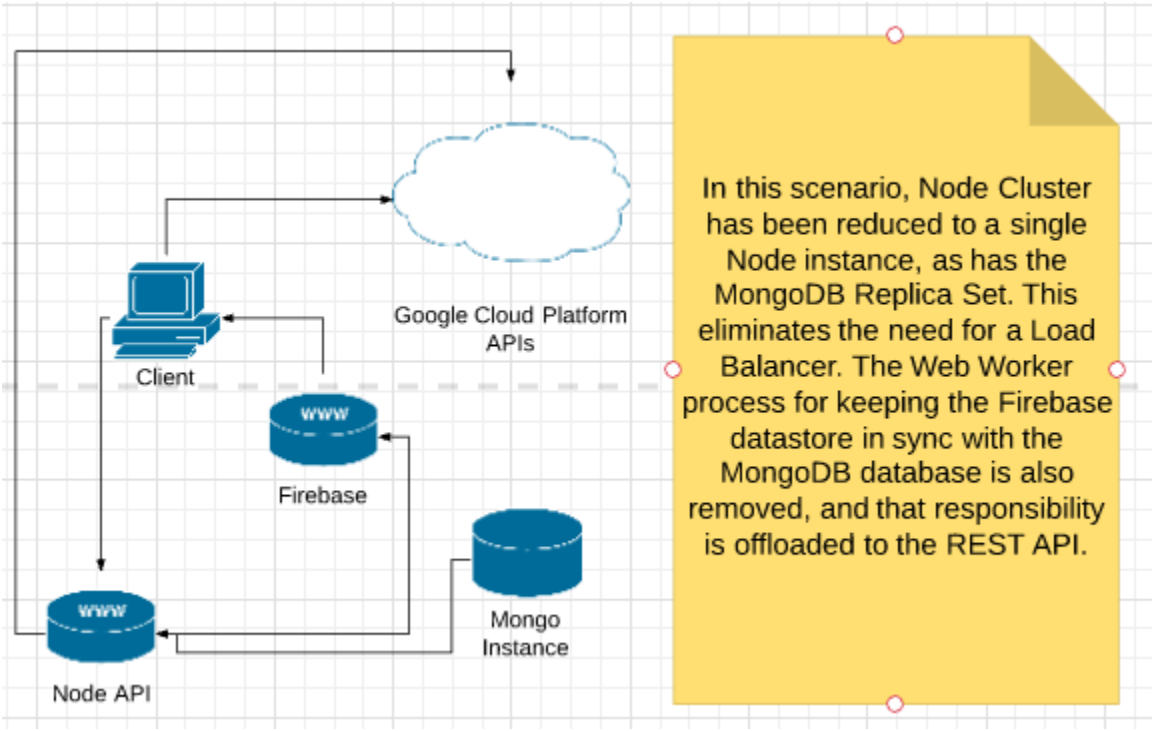


Figure 14 -- Infrastructure Diagram Four

After I had scaled my architecture down this far, I thought that perhaps I could scale it down even further. I did some further research into Firebase and Google Cloud Platform and saw that I could remove NodeJS and MongoDB from the landscape entirely, and instead run the application entirely on a GCP managed platform (Google Cloud (A), 2017). This would allow a more flexible migration path, as GCP provides a vast array of functionality ranging from Data Warehousing (Google Cloud (B), 2017) to Artificial Intelligence (Google Cloud (D), 2017). They also provide Application Server Hosting (Google Cloud (C), 2017), if it was decided to implement a custom server solution. In this case the architecture would look like below.

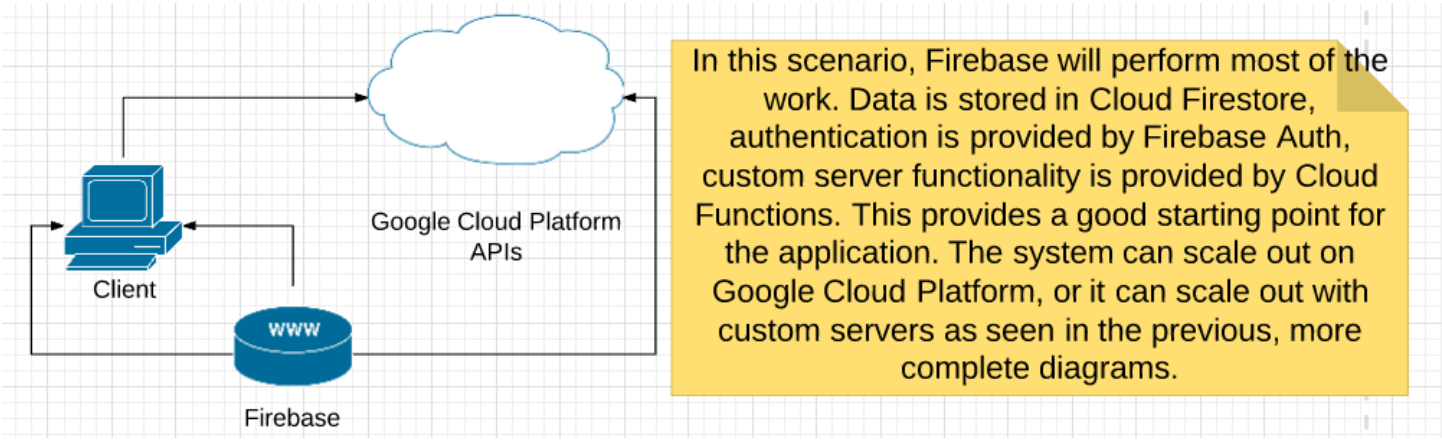
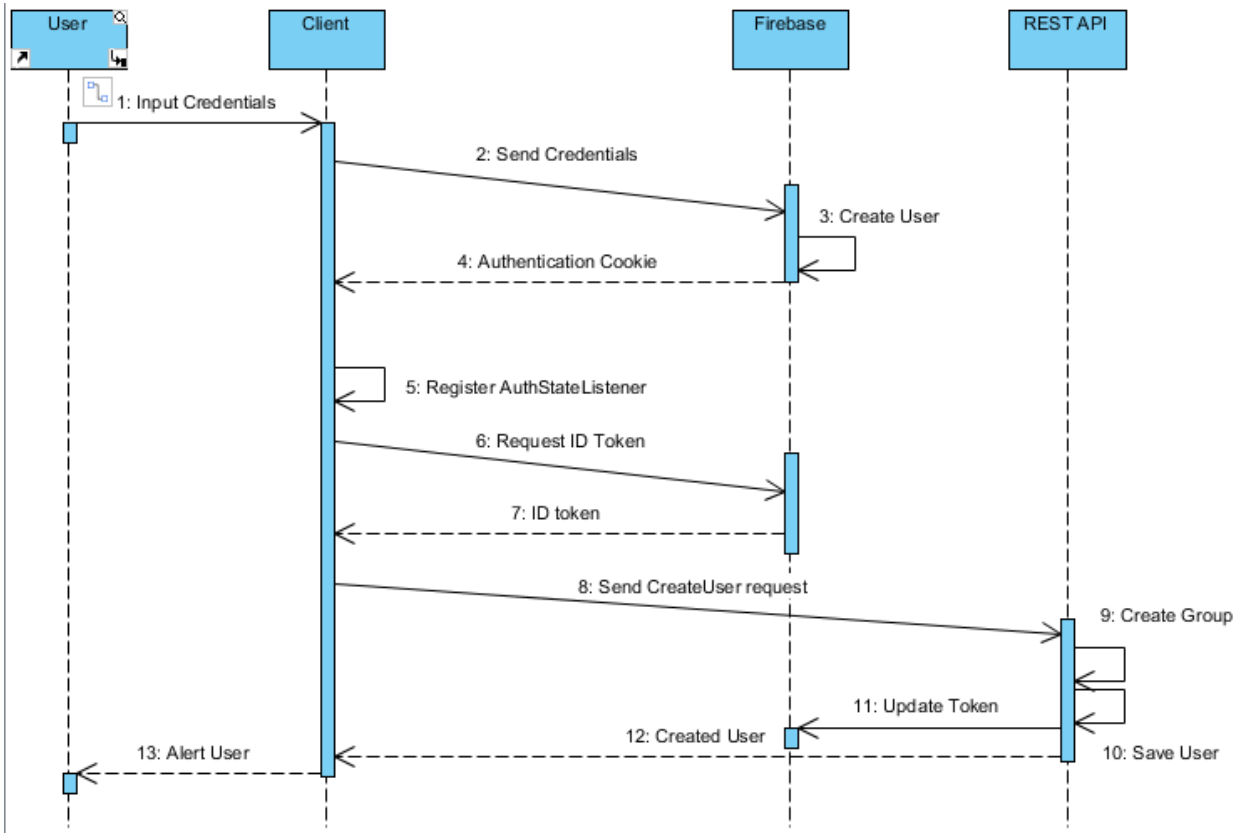


Figure 15 -- Infrastructure Diagram Five

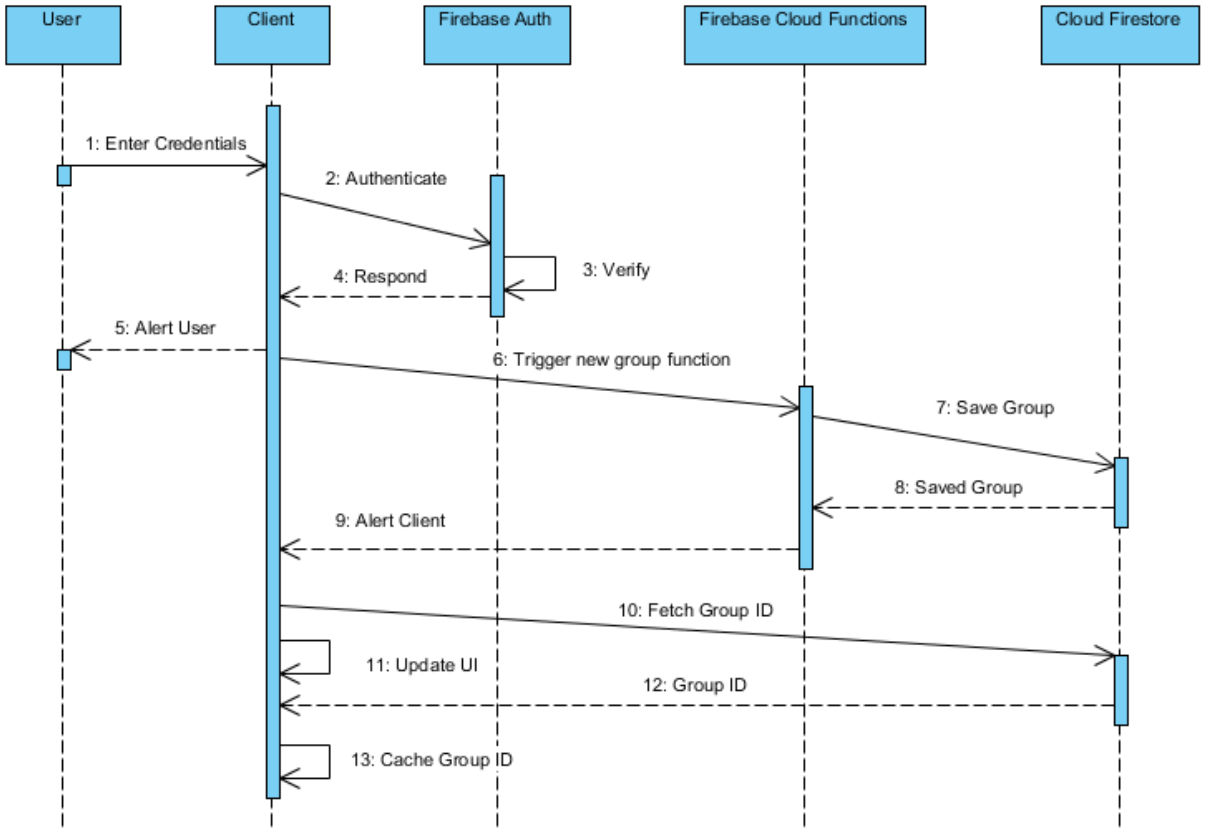
In the case above, the entire architecture outlined at the beginning of this section can be managed by Firebase and GCP. This would provide a much more scalable solution than implementing a custom landscape and managing multiple clusters of application servers.

To show how these architectures differ, I have included the sequence diagram showing how the initial sign up process would happen with first the REST API method, then with the Firebase method.



In this case, the authentication is handled by Firebase. The user inputs their credentials to the client. The client offloads the authentication process to Firebase. Firebase returns an authentication cookie which is used to track in login state on the client. The cookie also allows the client-side API to fetch an authentication token from firebase, which can be attached to the "Authorization" header for each request to the server. The server will validate the token and extract the user details. It will then create a new group and save the user from the details inferred in the token. It will then update the token in Firebase so that the group ID is available without the need for a DB read on each request. The client will now request a token, containing the groupID, from Firebase before each request to the server.

Figure 16 -- Sign Up REST API



In this scenario, it is assumed that the user who is signing up is attempting to create a new group. The user first inputs credentials. Firebase will authenticate, and return the response to the client. The client will make an asychronous call to trigger a Cloud Function, and then update the client UI and allow them to continue working. It would also be possible for the Firebase Auth module to trigger the cloud function, but this would provide less flexability than calling the function using REST. The cloud function will then save a new user group containing the firebase ID to Firestore. Once the Group has been saved, the client can request the Group ID for the currently logged in user. This value will be stored in the applications localCache for future requests. A check will be performed every time the application loads to see if the Group ID is cached. If not, it will be fetched from the server.

Figure 17 -- Sign Up GCP

In the second sequence diagram, it can be noted that Cloud Functions takes the place of the REST API, and Cloud Firestore takes the place of the MongoDB instance. This architecture is much closer to that of a *microservices* architecture, in that cloud functions are automatically scalable, single functions. The functions scale individually as needed, and are paid for as used (Firebase (G), 2017).

The main difference between the two architectures is how they handle repeat requests. In the first scenario, the Group ID is attached to the Firebase token. This is not an ideal practice, since the token will grow and must still be attached to the header, so each subsequent request will have that additional overhead. (Firebase (H), 2017). In the second version, a request is made to fetch the GroupID from Cloud Firestore, which is then stored in the Applications local state, and perhaps written to local storage also, to be kept between restarts.

DATA STRUCTURE

It is important to define the underlying data structures of the application. For the sake of brevity, this document will focus on the JSON representation of the data, rather than the SQL and Business Intelligence data models which are not on the roadmap for the foreseeable future.

Authentication

Initially the data model for authentication was based around storing the users inside of the user group collection. After deciding to use a more managed infrastructure approach with Firebase, the authentication data model was greatly reduced, as a lot of the details can be stored in the generic Firebase object.

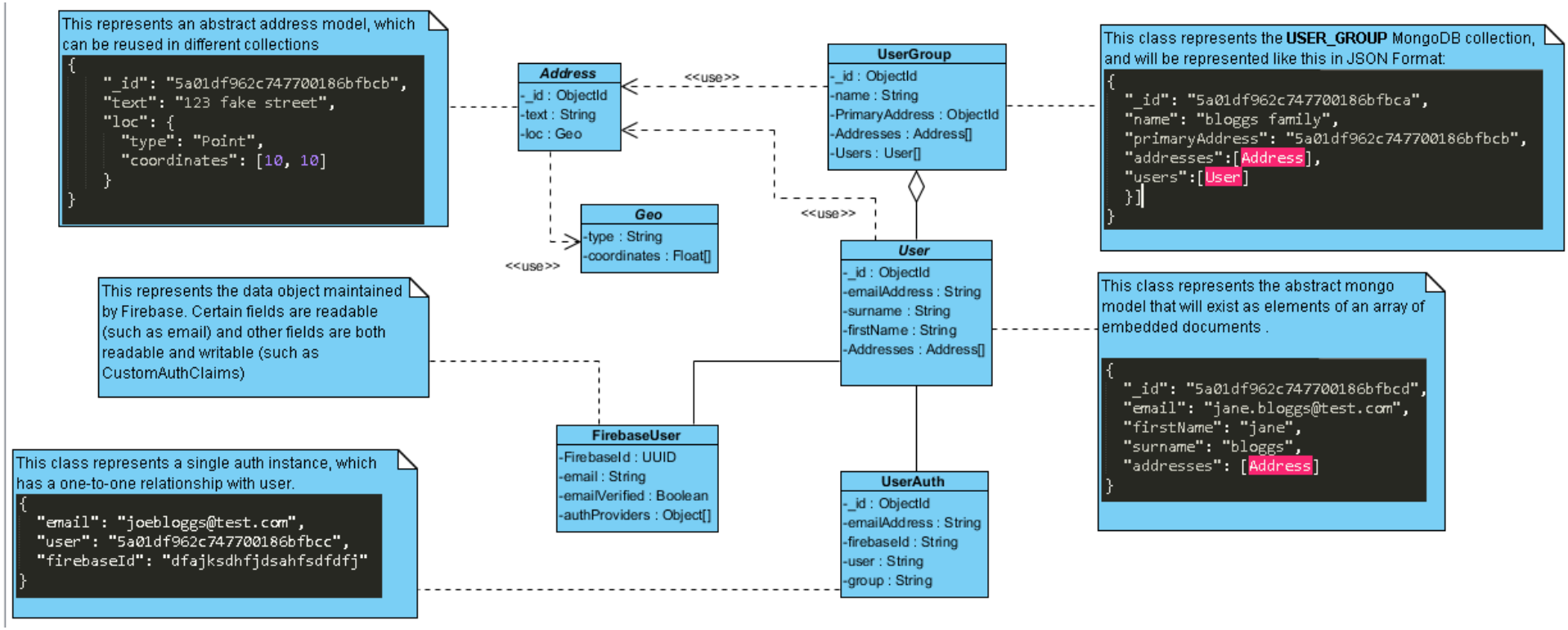


Figure 18 -- Authentication Data Model REST API

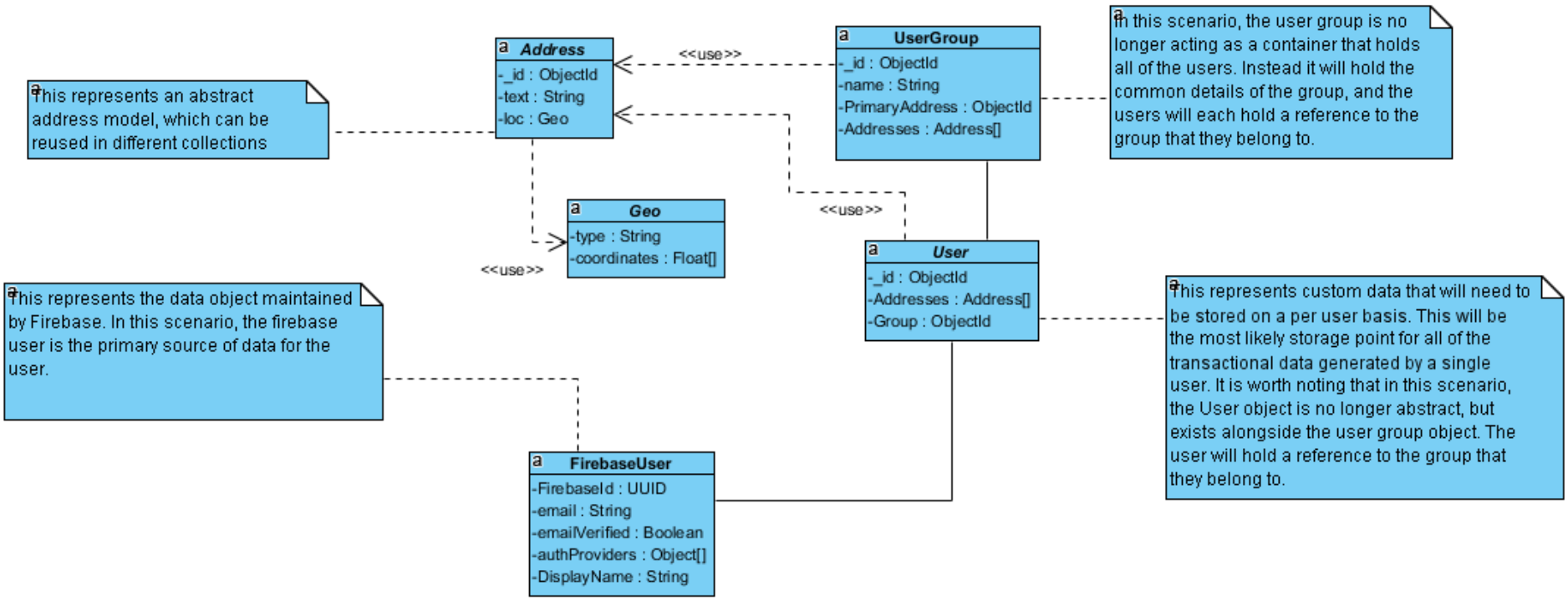


Figure 19 -- Authentication Data Model GCP

All other data models can be viewed in the appendix.

CLIENT-SIDE APPLICATION

For the client side there are several options. As I wished this application to be available to all potential users, it was important to choose a technology that was ubiquitous in modern life. For this reason, I chose only mobile and web applications for consideration for my client.

Native Mobile App	
Reference	(Techopedia, 2017)
Description	An application that is installed via the corresponding app store
Hybrid Mobile App	
Reference	(Bristowe, 2015)
Description	An application that is built using web technologies and ported to act as a native app
Single Page App	
Reference	(Wasson, 2013)
Description	A web application that only refreshes the relevant portion of the page on redirect
Progressive Web App	
Reference	(Google (J), 2017)
Description	A Web application that has some of the functionality of a native app on certain platforms

Figure 20 -- Possible Client Technologies

After reviewing the above technologies, I came up with the following advantages and disadvantages for each.

Native	
Description	Separate applications built for each platform (iOS, Android, Windows)
Pros	
	Full device access
	Most intuitive UI
	Data available offline
	Good user engagement
Cons	
	Must build separate applications for each platform
	Slow development
	Poor consistency
	Must pass through native app store
	Added complexity

Figure 21 -- Native App Pros/Cons

Hybrid	
Description	Application built as a Web Application, but packaged to work on as a native device
Pros	
	Built as a single application but runs like a native app on all platforms
	Faster development than separate native applications
	Data available offline
	Robust toolset in current JavaScript world
Cons	
	Does not provide as good of a native experience as a native application
	Must pass through native app store
	Does not have as many privileges on device as a native application

Figure 22 -- Hybrid App Pros/Cons

Single Page Application	
Description	Javascript Web Application that is executed in the browser
Pros	
	Runs in browser
	No need to be published on app store
	Discoverable via search engine
	Very fast development time
	Robust toolset in current JavaScript world
Cons	
	Virtually no device access
	Data not available offline
	Application is unavailable while offline

Figure 23 -- Single Page App Pros/Cons

Progressive Web Application	
Description	Web application that takes advantage of Service Workers to act more like a native application
Pros	
	Runs in browser
	Available offline (where Service Workers are available)
	Installable (where Service Workers are available)
	Discoverable via search engine
	No need to be published on app store
	Very fast development time
	Robust toolset in current JavaScript world
Cons	
	Service Workers not supported everywhere (feature in development as of iOS 11)
	Does not provide as good of a native experience as a native application
	Emerging technology
	Does not have as many privileges on device as a native application

Figure 24 -- Progressive Web App Pros/Cons

My final decision was to go with the Progressive Web Application, as it provided the best balance between the features of a native application and the availability of a web application.

Final Chosen Technologies

My final chosen technology stack is to be a Progressive Web Application, hosted on Firebase. This application will communicate with a Cloud Firestore database and Cloud Functions serverless REST API. This will facilitate a highly scalable system that should also be highly available.

Conclusion

During the process of designing this system, I feel like I learned a lot about the various technologies that are available to build web applications. I was surprised at the amount of research that had to be completed to flesh out a full technology stack.

I also felt that I learned a lot about how the requirements of a system can change. Early on, I had anticipated building a monolithic server that would handle all data access for a client-side app. After doing my research on serverless architectures, I decided to switch to a more managed infrastructure approach. I feel that this was excellent experience in building a more real-to-life application than what I had done previously.

Appendix

FIGURES

Infrastructure

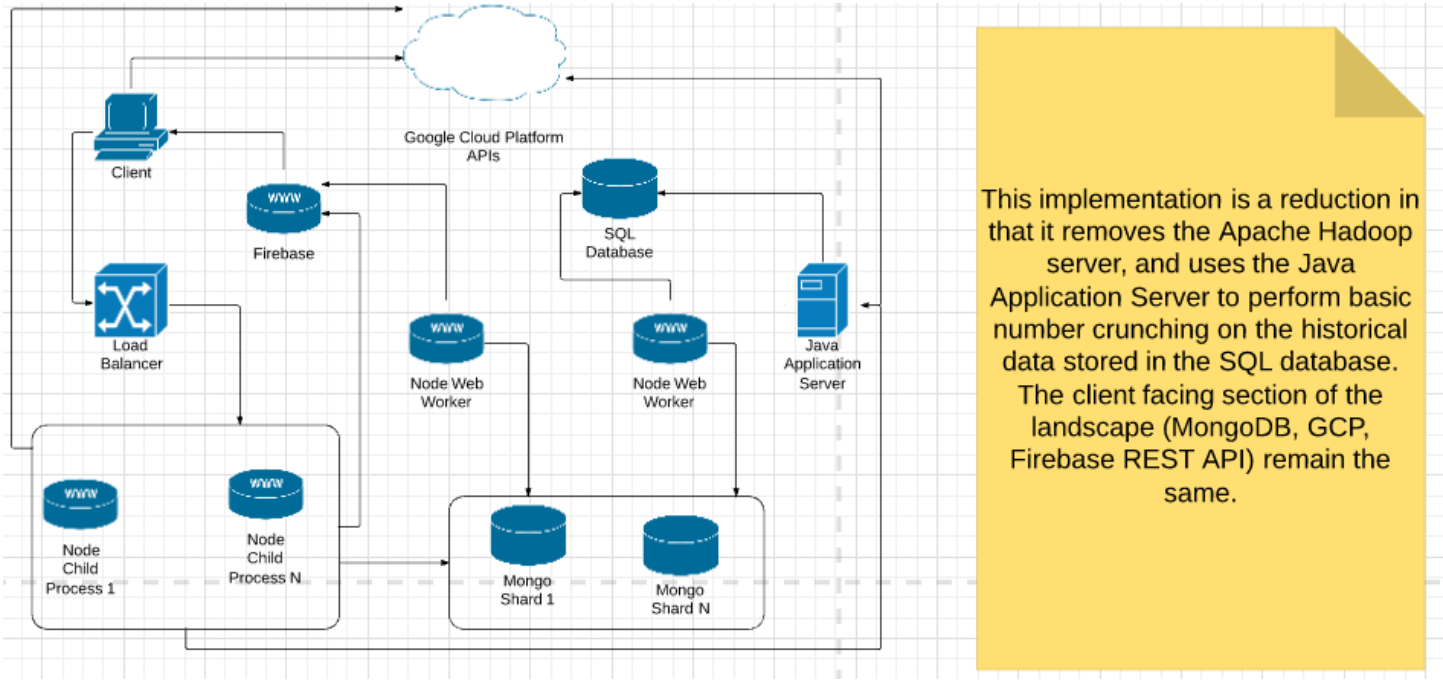


Figure 25 -- Infrastructure Diagram Two

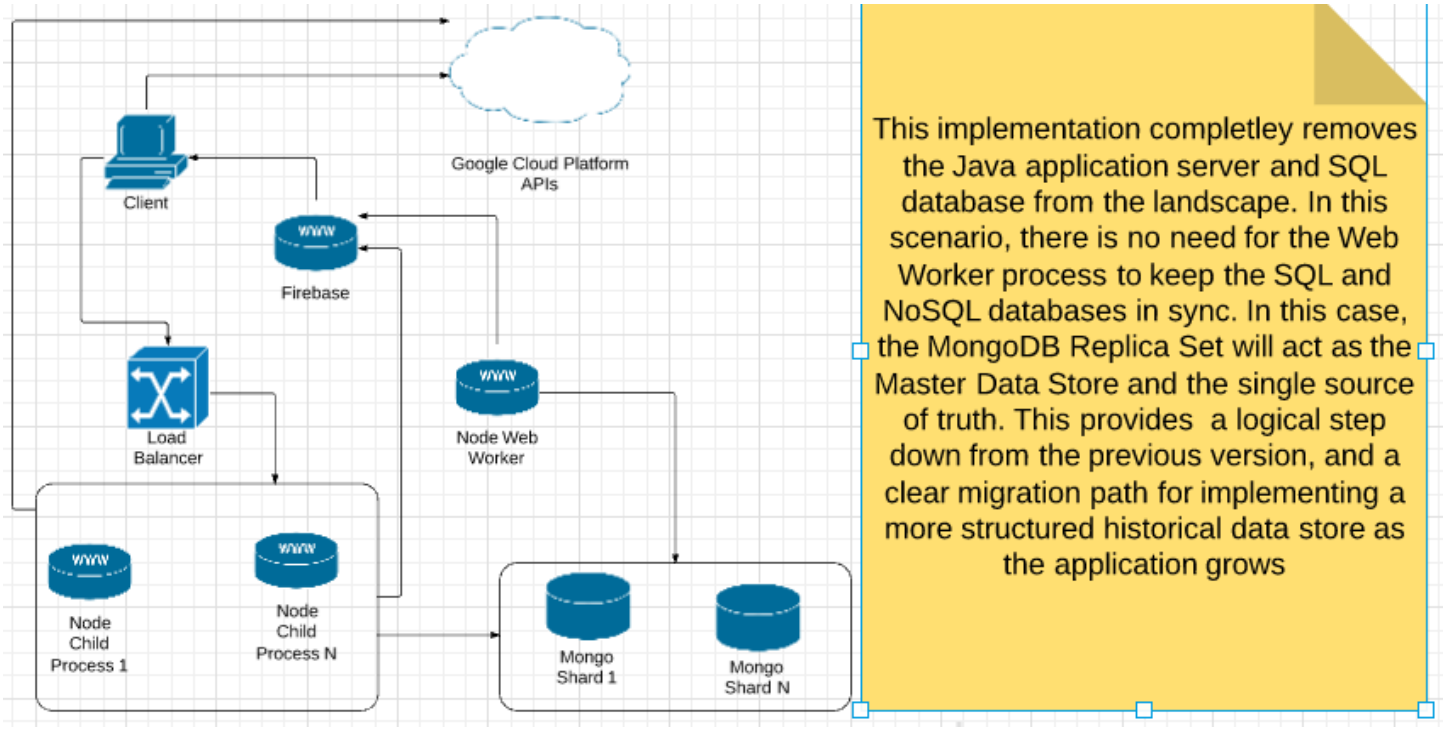


Figure 26 -- Infrastructure Diagram Three

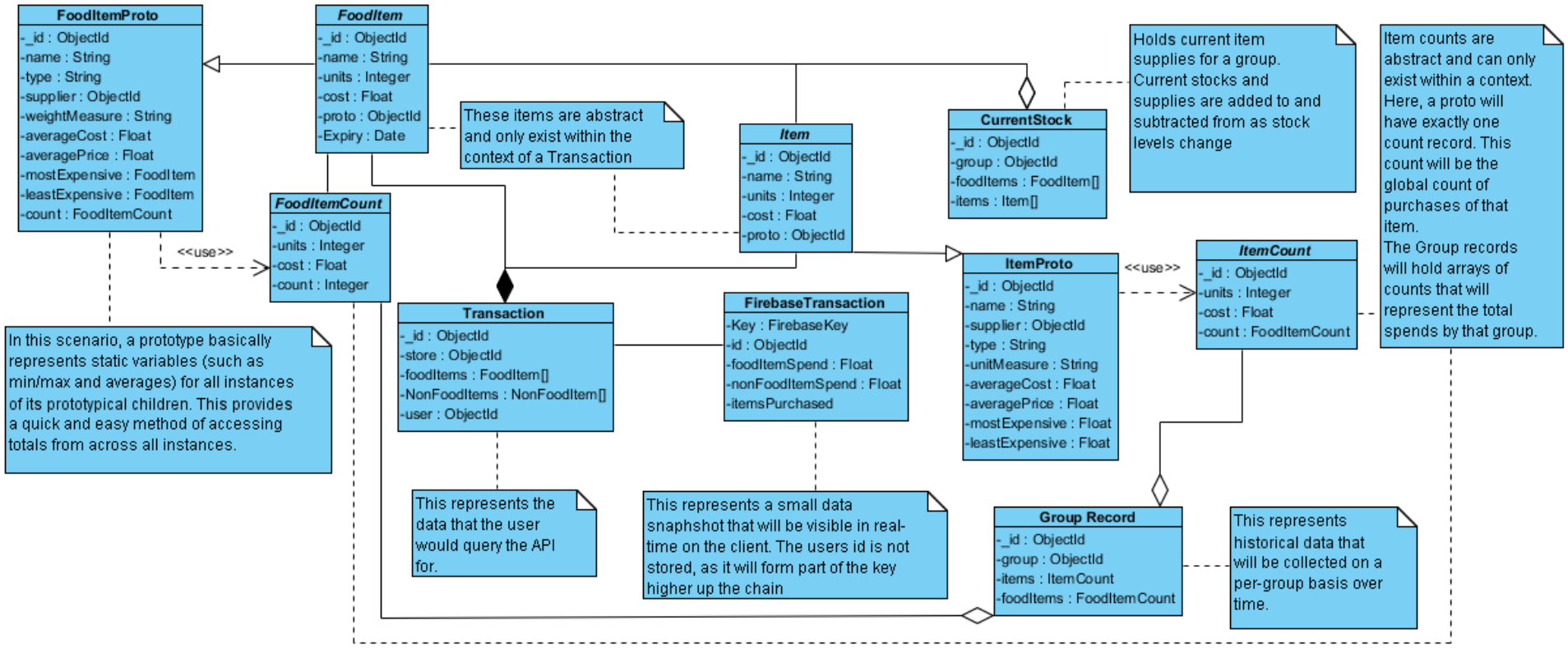


Figure 27 -- Transaction Data Model

Use Cases

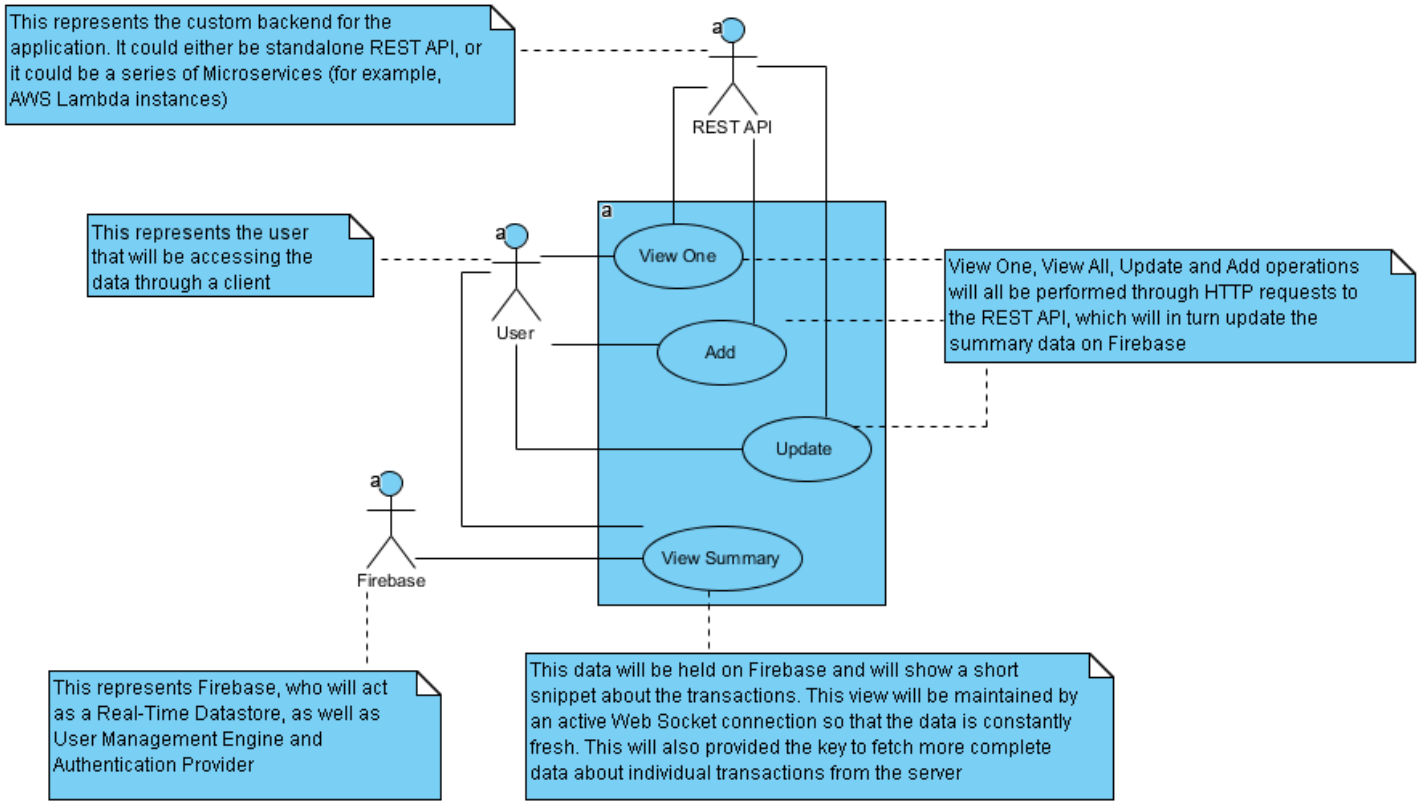


Figure 28 -- Transaction Use Case

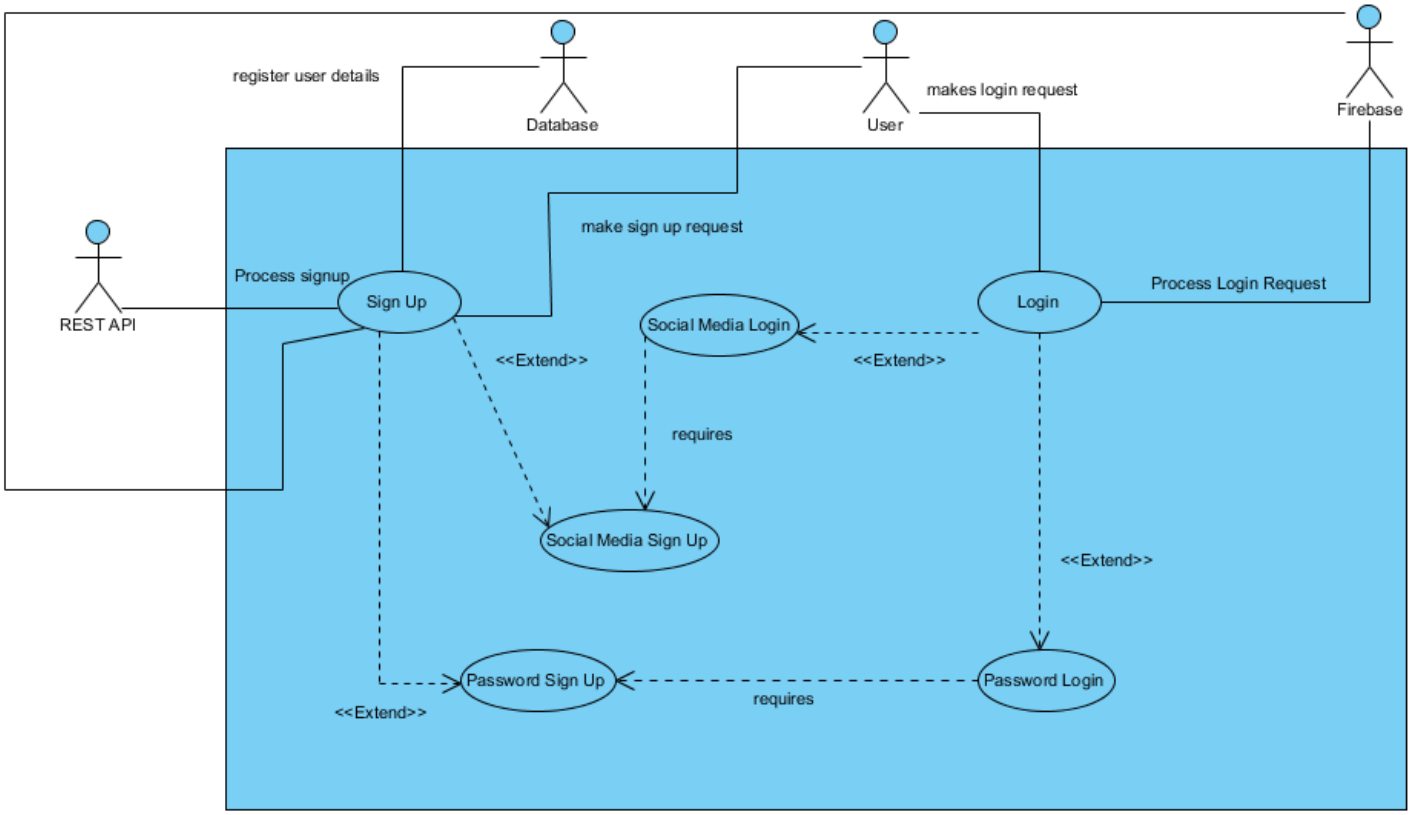


Figure 29 -- Authentication Use Case

Sequence Diagrams

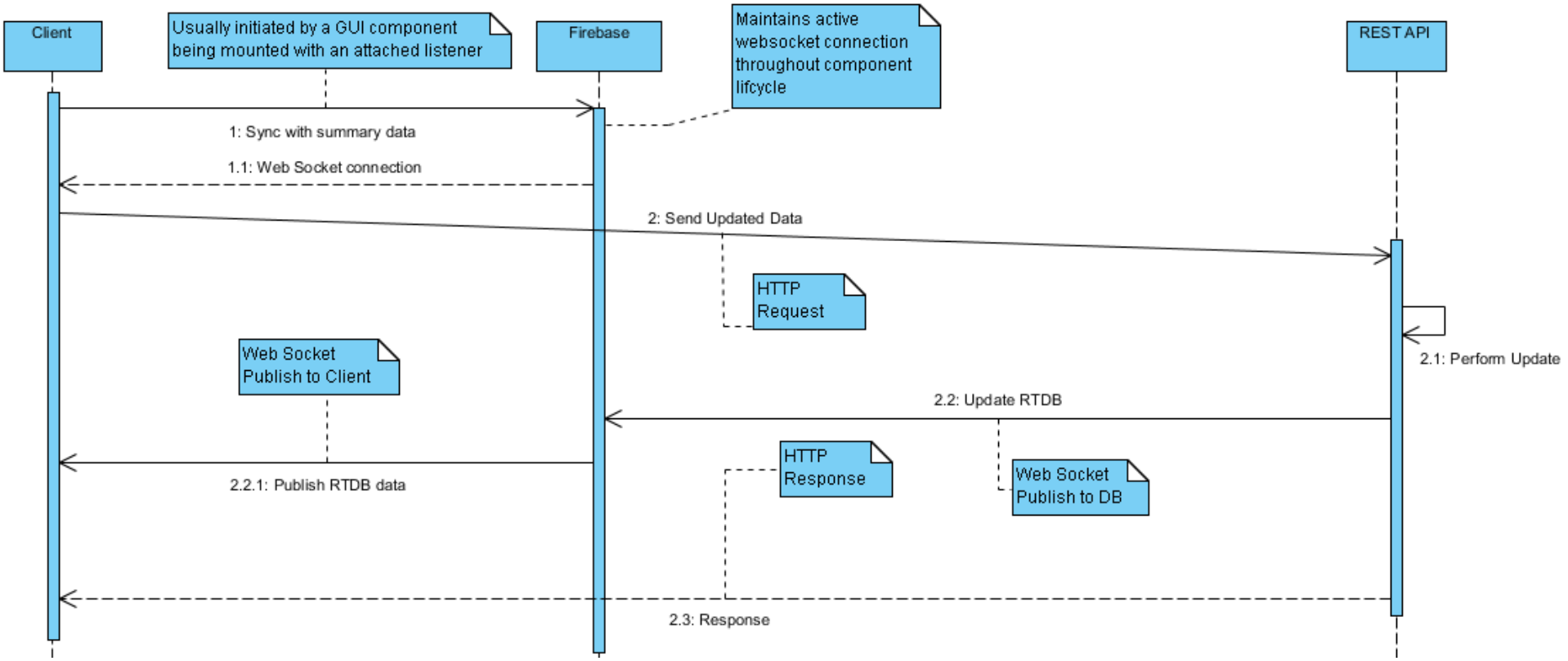


Figure 30 -- REST API Basic Data Transfer

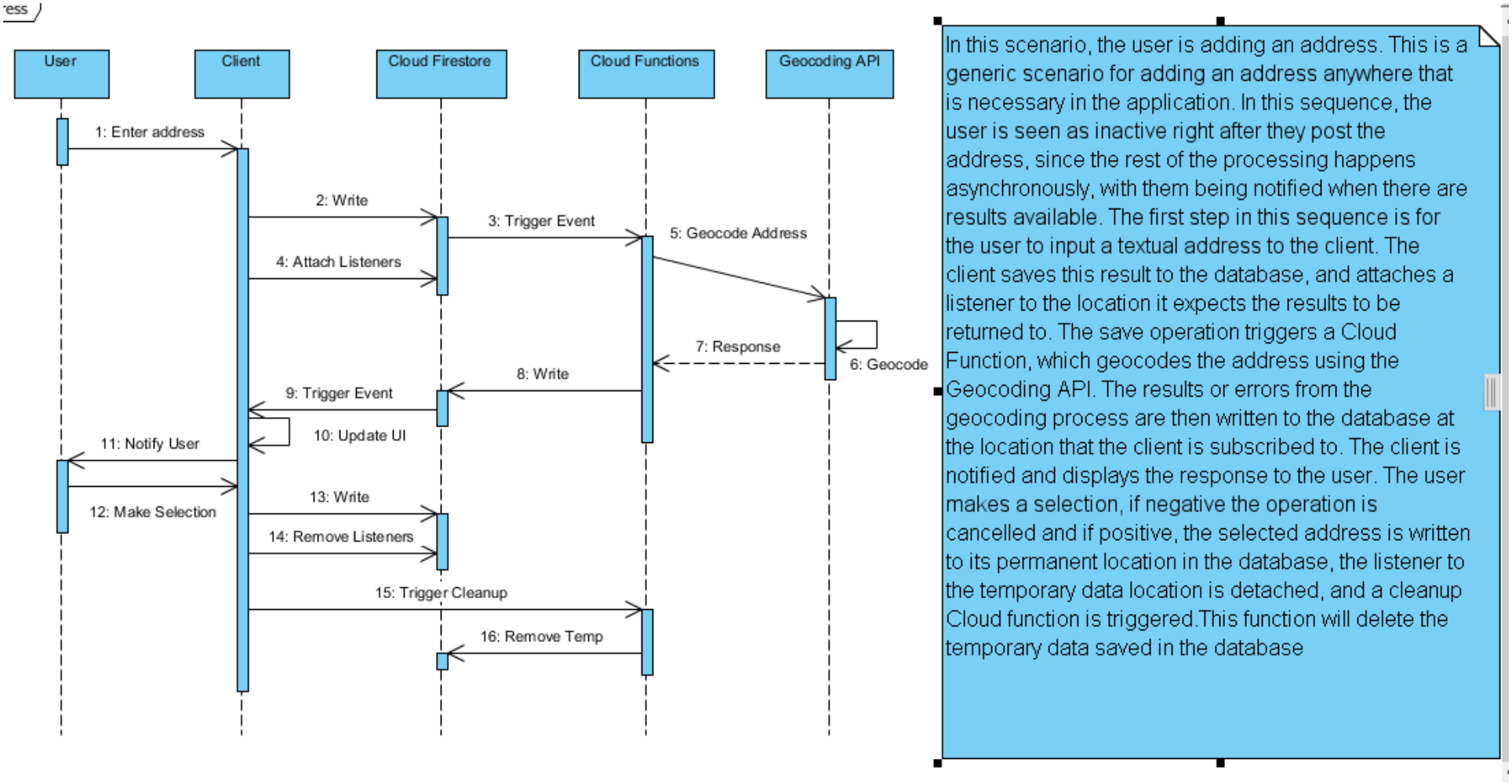


Figure 31 -- GCP Geocode Address Sequence

Activity Diagrams

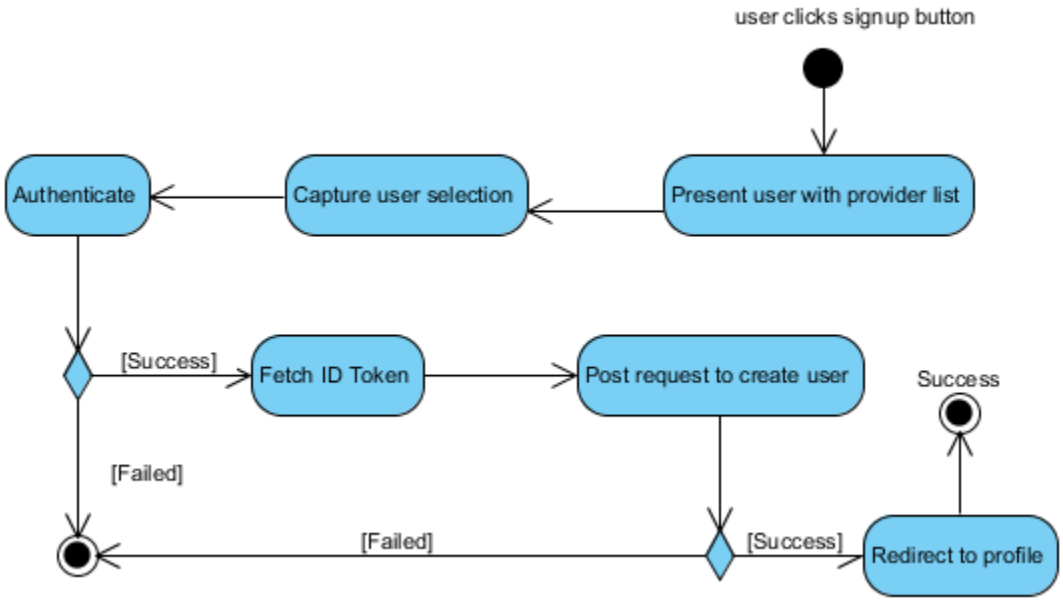


Figure 32 -- Client Side Sign Up REST

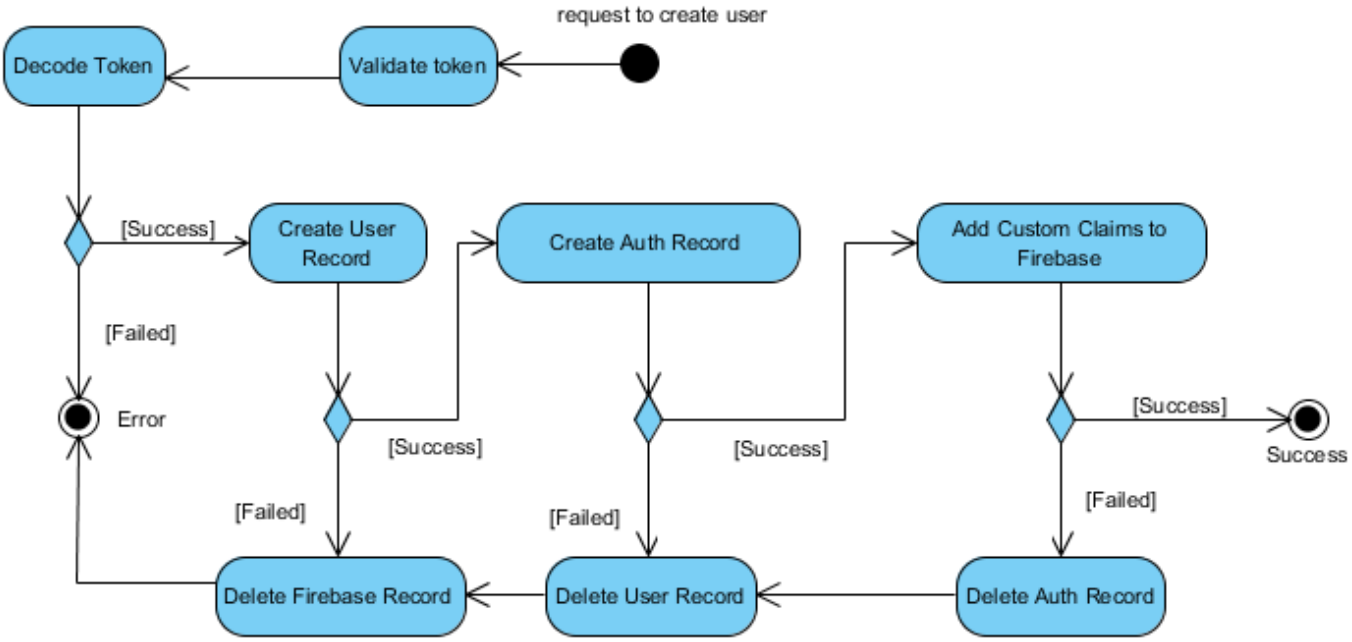


Figure 33 -- Server-Side Sign Up REST

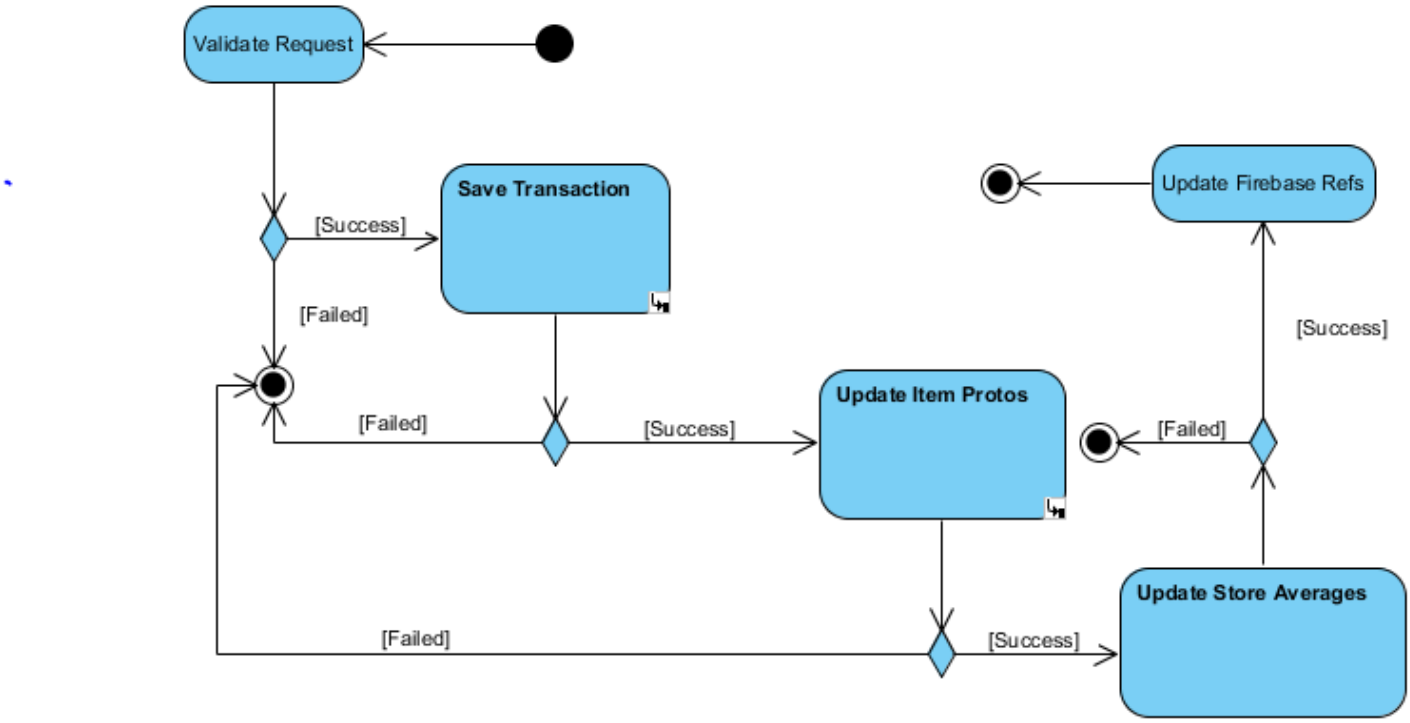


Figure 34 -- High Level Add Transaction REST Activity

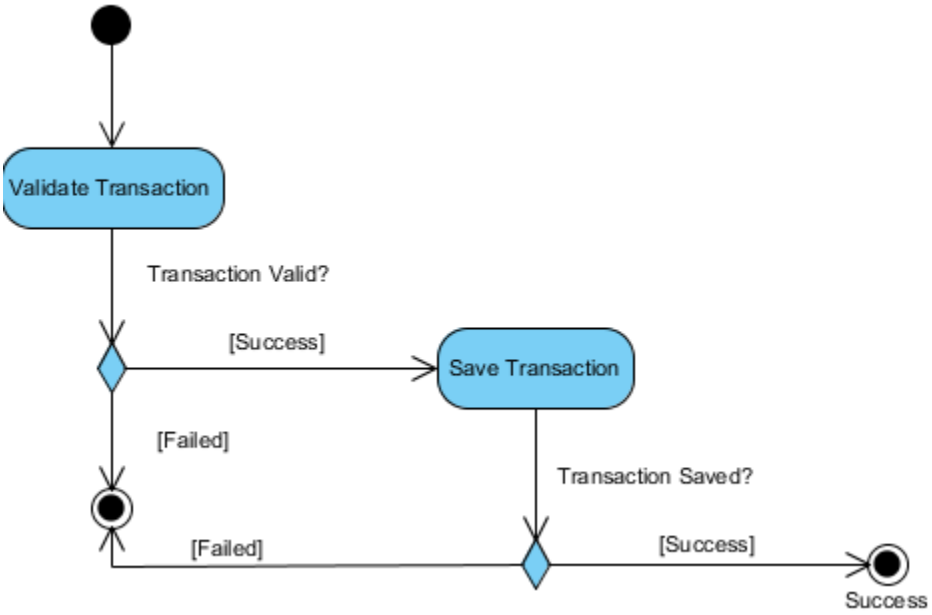


Figure 35 -- Save Transaction REST Activity

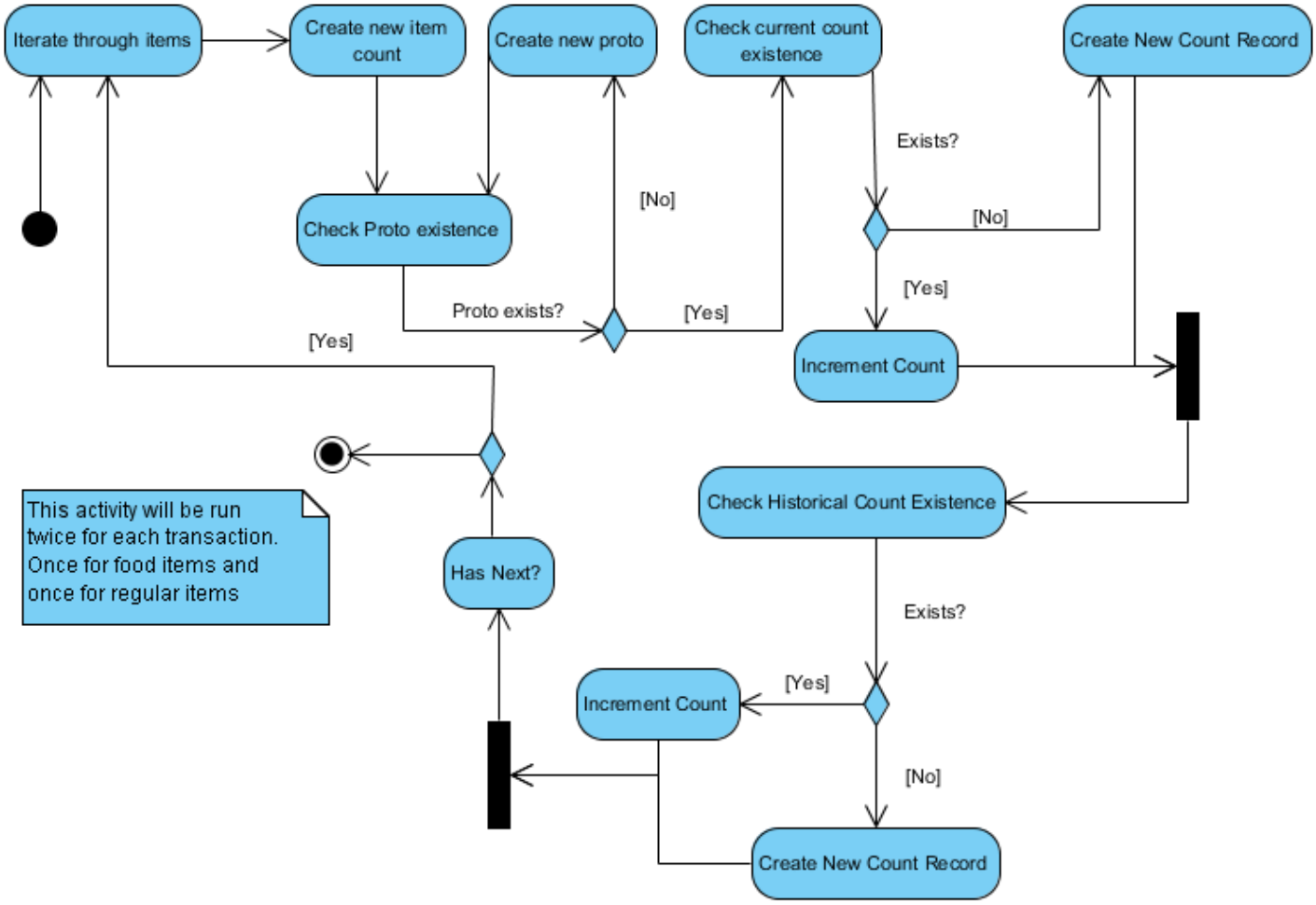


Figure 36 -- Update Item Protos REST Activity

REFERENCES

Algolia, 2017. *Search made Simple.* [Online]
Available at: https://www.algolia.com/
[Accessed 28 10 2017].

Ambler, S., 2016. *The Disiplined Agile Framework -- The Principals of Lean Software Development.* [Online]
Available at: http://www.disciplinedagiledelivery.com/lean-principles/
[Accessed 03 11 2017].

AWS (A), 2017. *Start Building on AWS Today.* [Online]
Available at: https://aws.amazon.com/
[Accessed 29 10 2017].

AWS (B), 2017. *AWS -- EC2.* [Online]
Available at: https://aws.amazon.com/ec2/?nc2=h_m1
[Accessed 29 10 2017].

AWS (C), 2017. *AWS -- Lambda.* [Online]
Available at: https://aws.amazon.com/lambda/?nc2=h_m1
[Accessed 29 10 2017].

AWS (D), 2017. *AWS -- Lex.* [Online]
Available at: https://aws.amazon.com/lex/?nc2=h_a1
[Accessed 29 10 2017].

AWS (E), 2017. *AWS -- API Gateway.* [Online]
Available at: https://aws.amazon.com/api-gateway/?nc2=h_m1
[Accessed 29 10 2017].

AWS (F), 2017. *AWS -- Aurora.* [Online]
Available at: https://aws.amazon.com/rds/aurora/?nc2=h_m1
[Accessed 29 10 2017].

AWS (G), 2017. *AWS -- DynamoDB.* [Online]
Available at: https://aws.amazon.com/dynamodb/?nc2=h_m1
[Accessed 29 10 2017].

AWS (H), 2017. *AWS -- S3.* [Online]
Available at: https://aws.amazon.com/s3/?nc2=h_m1
[Accessed 29 10 2017].

Bristowe, J., 2015. *What is a Hybrid Mobile App.* [Online]
Available at: https://developer.telerik.com/featured/what-is-a-hybrid-mobile-app/
[Accessed 02 11 2017].

Capgemini, 2017. *Big and Fast Data: The Rise of Insight Driven Business.* [Online]
Available at: https://www.capgemini.com/wp-content/uploads/2017/07/big_fast_data_the_rise_of_insight-driven_business-report.pdf
[Accessed 02 11 2017].

ExpertsExchange, 2014. *Waterfall Model Pros and Cons.* [Online]
Available at: https://www.experts-exchange.com/articles/17441/Waterfall-Model-pros-and-cons.html
[Accessed 02 11 2017].

EY, 2016. *Global Insurance CFO Survey 2016.* [Online]
Available at: http://www.ey.com/Publication/vwLUAssets/Global_Insurance_CFO_Survey_2016/$FILE/EY-global-insurance-cfo-survey-2016.pdf
[Accessed 5 10 2017].

Firebase (B), 2017. *Firebase Web Authentication.* [Online]
Available at: https://firebase.google.com/docs/auth/web/start
[Accessed 04 11 2017].

Firebase (A), 2017. *Firebase.* [Online]
Available at: https://firebase.google.com/
[Accessed 04 11 2017].

Firebase (C), 2017. *Firebase Server Authentication.* [Online]
Available at: https://firebase.google.com/docs/auth/admin/
[Accessed 02 11 2017].

Firebase (D), 2017. *Firebase Cloud Functions Use Cases.* [Online]
Available at: https://firebase.google.com/docs/functions/use-cases
[Accessed 03 11 2017].

Firebase (E), 2017. *Firebase Real Time Database.* [Online]
Available at: https://firebase.google.com/docs/database/web/start
[Accessed 04 11 2017].

Firebase (F), 2017. *Firebase Cloud Firestore.* [Online]
Available at: https://firebase.google.com/docs/firestore/quickstart
[Accessed 03 11 2017].

Firebase (G), 2017. *Firebase Cloud Functions.* [Online]
Available at: https://firebase.google.com/docs/functions/
[Accessed 25 10 2017].

Firebase (H), 2017. *Firebase ID tokens best practices.* [Online]
Available at: https://firebase.google.com/docs/auth/admin/custom-claims#best_practices_for_custom_claims
[Accessed 10 25 2017].

Firebase (I), 2017. *Firebase Cloud Messaging.* [Online]
Available at: https://firebase.google.com/docs/cloud-messaging/
[Accessed 25 10 2017].

Firebase (J), 2017. *Firebase Storage.* [Online]
Available at: https://firebase.google.com/docs/storage/
[Accessed 25 10 2017`].

Firebase (K), 2017. *Firebase Hosting.* [Online]
Available at: https://firebase.google.com/docs/hosting/
[Accessed 25 10 2017].

Gartner, 2017. *Gartner Says Worldwide IT Spending Forecast to Grow 2.4 Percent in 2017.* [Online]
Available at: https://www.gartner.com/newsroom/id/3759763
[Accessed 02 11 2017].

Google (H), 2017. *GCP -- Maps UI.* [Online]
Available at: https://developers.google.com/maps/documentation/javascript/?hl=en_US
[Accessed 25 10 2017].

Google (J), 2017. *Progressive Web Apps.* [Online]
Available at: https://developers.google.com/web/progressive-web-apps/
[Accessed 02 11 2017].

Google Cloud (A), 2017. *Google Cloud Platform.* [Online]
Available at: https://cloud.google.com/
[Accessed 25 10 2017].

Google Cloud (B), 2017. *GCP -- BigQuery.* [Online]
Available at: https://cloud.google.com/bigquery/
[Accessed 25 10 2017].

Google Cloud (C), 2017. *GCP -- AppEngine.* [Online]
Available at: https://cloud.google.com/appengine/
[Accessed 25 10 2017].

Google Cloud (D), 2017. *GCP -- MachineLearning.* [Online]
Available at: https://cloud.google.com/products/machine-learning/
[Accessed 25 10 2017].

Google Cloud (E), 2017. *GCP -- Maps Geocoding.* [Online]
Available at: https://developers.google.com/maps/documentation/geocoding/intro?hl=en_US
[Accessed 25 10 2017].

Google Cloud (F), 2017. *GCP -- Analytics.* [Online]
Available at: https://developers.google.com/analytics/devguides/config/?csw=1
[Accessed 25 10 2017].

Google Cloud (G), 2017. *GCP -- Cloud SQL.* [Online]
Available at: https://cloud.google.com/sql/docs/
[Accessed 25 10 2017].

Google Cloud (I), 2017. *GCP -- Maps Places.* [Online]
Available at: https://developers.google.com/places/web-service/?hl=en_US
[Accessed 15 10 2017].

Heroku, 2017. *Welcome to Heroku.* [Online]
Available at: https://www.heroku.com/
[Accessed 28 10 2017].

Kumar, V., 2017. *Analytics Week -- Big Data Facts.* [Online]
Available at: https://analyticsweek.com/content/big-data-facts/
[Accessed 02 11 2017].

LEANKit, 2017. *What is Kanban.* [Online]
Available at: https://leankit.com/learn/kanban/what-is-kanban/
[Accessed 02 11 2017].

mLab, 2017. *2017.* [Online]
Available at: https://mlab.com/
[Accessed 28 10 2017].

MongoDB Atlas, 2017. *MongoDB Atlas.* [Online]
Available at: https://www.mongodb.com/cloud/atlas/pricing
[Accessed 28 10 2017].

Nagy, J., 2015. *JSTechDesign s-- UX: Dont be too rigid.* [Online]
Available at: https://www.jstechdesigns.com/Blog/ux-dont-be-too-rigid
[Accessed 02 11 2017].

Pittet, S., 2017. *Atlassian -- Continuous integration vs. continuous delivery vs. continuous deployment.* [Online]
Available at: https://www.atlassian.com/continuous-delivery/ci-vs-ci-vs-cd
[Accessed 03 11 2017].

ScrumGuides, 2016. *The Scrum Guide.* [Online]
Available at: http://www.scrumguides.org/scrum-guide.html
[Accessed 02 11 2017].

Surge, 2017. *Static web publishing.* [Online]
Available at: https://surge.sh/
[Accessed 28 10 2017].

Techopedia, 2017. *Native Mobile App.* [Online]
Available at: https://www.techopedia.com/definition/27568/native-mobile-app
[Accessed 05 11 2017].

TheLeanStartup, 2017. *The Lean Startup -- Methodology.* [Online]
Available at: http://theleanstartup.com/principles
[Accessed 04 11 2017].

Wasson, M., 2013. *ASP.NET - Single-Page Applications: Build Modern, Responsive Web Apps with ASP.NET.* [Online]
Available at: https://msdn.microsoft.com/en-us/magazine/dn463786.aspx
[Accessed 02 11 2017].

Weigers, K. E., 2003. *Software Requirements.* s.l.:Microsoft.

Wilcox, R., 2017. *Toptal -- Your Boss Won't Appreciate TDD: Try This Behavior-Driven Development Example.* [Online]
Available at: https://www.toptal.com/freelance/your-boss-won-t-appreciate-tdd-try-bdd
[Accessed 03 11 2017].

Zeit, 2017. *Now.sh.* [Online]
Available at: https://zeit.co/now
[Accessed 28 10 2017].