Home Resource Planner

Full Design Document

Joe Wemyss | Software Systems Development | September 2017

Contents

[Project Goal 2](#_Toc497643955)

[Motivation 2](#_Toc497643956)

[Goal 2](#_Toc497643957)

[User Analysis 3](#_Toc497643958)

[Target Audience 3](#_Toc497643959)

[End User Functionality 3](#_Toc497643960)

[Methodologies 5](#_Toc497643961)

[Requirements Analysis 6](#_Toc497643962)

[Functional Areas 6](#_Toc497643963)

[Login 6](#_Toc497643964)

[Track Suppliers 6](#_Toc497643965)

[Track Finance 7](#_Toc497643966)

[Inventory Management 7](#_Toc497643967)

[Functional Requirements 9](#_Toc497643968)

[Architectural Overview 10](#_Toc497643969)

[Basic Non-Functional Requirments 10](#_Toc497643970)

[Application Structure 10](#_Toc497643971)

[Appendix 15](#_Toc497643972)

[References 19](#_Toc497643973)

Figures

[Figure 1--Auth Use Case 12](#_Toc496570059)

[Figure 2--Conceptual Module Overview 13](#_Toc496570060)

[Figure 3--Full Physical Implementation 13](#_Toc496570061)

[Figure 4--Mid Physical Implementation 14](#_Toc496570062)

[Figure 5--Small Physical Implementation 15](#_Toc496570063)

# 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)([Kumar, 2017]( https://github.com/JavaTheNutt/final\_year\_project\_documentation/blob/master/docs/design\_docs/README.md#kumar-v-2017))). The amount money being spent by companies to manage this data is also growing ( (Gartner, 2017)([Gartner, 2017]( https://github.com/JavaTheNutt/final\_year\_project\_documentation/tree/master/docs/design\_docs#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)([Nagy, 2015]( https://github.com/JavaTheNutt/final\_year\_project\_documentation/blob/master/docs/design\_docs/README.md#nagy-j-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.

## End User Functionality

For the most part, functionality will be delivered iteratively as requirements change. As such, functionality can be thought of as being dynamic throughout the lifecycle of the application. The initial core user functionality will be the ability to create an account and manage users within this account. This is one of the most important steps in the application, as this is where all the authentication must be set up.

The next “package” of functionality will be the ability to add suppliers, with geospatial addresses. There should be some type of predictive technology when entering supplier details, based on anonymised data that other people have entered before, such as if you type in “Aldi” it will show a list of Aldi stores nearby based on your location and data that people have entered before.

The next “package” of functionality will be the ability to track transactions. These transactions will be linked to either physical suppliers like were entered in the last step, or to “virtual” suppliers such as utilities providers. Aggregation will be performed on these transactions and will be available to view in various format. One format that I hope to implement is that a user could mouse over a location on the map and see their total spend in that location, as well as some anonymised averages from the wider community.

This would provide the central core functionality to the system, which could be considered the Minimum Viable Product(MVP). This is my minimum aim for the final iteration of the project.

The next “package” of functionality to be implemented would be the ability to add details about food in the home. This would be a separate piece of functionality to the financial section, except that the food purchased would be purchased as part of a transaction that had been recorded in the financial section. The difficulty here is that I am unaware of technologies that could extract the details from a receipt, so this feature would require a lot of manual data entry that must be minimised as much as possible until a more user-friendly manner can be found.

The next “package” would be a shopping list function, that users could create a digital shopping lists that could be prepopulated with common items and tailored for particular shopping trips.

The final “package” of possible functionality that I have considered is the ability to create recipes. These recipes would decrement the current supplies of food, while also allowing meal planning for the future.

It goes without saying that each of these feature-sets would also be feeding back into one another, such as a recipe calculating the costs of its ingredients while subtracting the stocks of food already available, and inserting required ingredients into the shopping lists. There should also be some form of a dashboard interface that would show totals from throughout the dataset.

It is highly unlikely that there will be time to implement all of these feature sets, but it is good to have future releases planned to help build a roadmap for the future.

# 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)([ExpertsExchange, 2014]( https://github.com/JavaTheNutt/final\_year\_project\_documentation/blob/master/docs/design\_docs/README.md#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)([ScrumGuides, 2016]( https://github.com/JavaTheNutt/final\_year\_project\_documentation/blob/master/docs/design\_docs/README.md#expertsexchange-2014)) and Kanban (LEANKit, 2017)([LEANKit, 2017]( https://github.com/JavaTheNutt/final\_year\_project\_documentation/blob/master/docs/design\_docs/README.md#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) (Wilcox, 2017)([Wilcox, 2017]( https://github.com/JavaTheNutt/final\_year\_project\_documentation/blob/master/docs/design\_docs/README.md#wilcox-r-2017)), LEAN (Ambler, 2016)([Ambler, 2016]( https://github.com/JavaTheNutt/final\_year\_project\_documentation/blob/master/docs/design\_docs/README.md#ambler-s-2016)), CI/CD (Pittet, 2017)([Pittet, 2017]( https://github.com/JavaTheNutt/final\_year\_project\_documentation/blob/master/docs/design\_docs/README.md#pittet-s-2017)).

The prototype will hopefully be able to serve as a LEAN Minimum Viable Product(MVP) (TheLeanStartup, 2017)([TheLeanStartup, 2017]( https://github.com/JavaTheNutt/final\_year\_project\_documentation/blob/master/docs/design\_docs/README.md#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)([Weigers, 2003](https://github.com/JavaTheNutt/final\_year\_project\_documentation/blob/master/docs/design\_docs/README.md#weigers-k-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 to for consideration:

* Streamlined login through third party SSO providers
  + Should cater to the major players in this field, such as Google, Facebook, Github
* Transactions
  + CRUD functionality
  + Made up of Items
  + Has aggregate totals
  + Provide mainly numerical data
  + Child relationship with Supplier
* Suppliers
  + CRUD functionality
  + May be physical or virtual
  + If physical, should have geospatial coordinates
  + May also be service providers
  + Provide mainly categorical data
  + Provide a source to generate Transactions
* Inventory
  + CRUD functionality
  + Should track turnover
  + Should provide usage levels for Budget
  + Fast changing data
  + Needs to be dynamic
* 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
* Correlation between functional areas
* Dashboard for showing aggregated data from across functional areas.
  + Should be customisable
    - User should be able to reorder items on page to show what they feel is most relevant to them.
    - Customisability state should be stored in the URL somehow, so that users can “bookmark” favoured data representations
  + Should present a graphical representation of the data.

# Architectural Overview

I decided that the best way to design a system architecture was from the outside in. I decided to first design the system landscape at scale, and then prune it back to determine what the Minimum Viable Product could be.

## Basic Non-Functional Requirments

The first step in implementing an architectural plan is to define some very basic non-functional requirements. An example of this is security. Whatever architectural solution is chosen must be secure, but must also be simple for a user, such as incorporating third party auth providers, such as Google or Facebook.

Another non-functional requirement that must be considered is performance. There are a couple of items that must be considered here. One of these is managing multiple concurrent users. This is a functional requirement that will possibly have a trade-off against another performance KPI, performant data processing. The reason for this is that languages that provide excellent concurrency controls, tend to be subpar at data processing, and vice versa. The example that I would use here is the merits of Java vs. NodeJS.

Another performance aspect that must be considered is how to optimise large network requests. This will need to be carefully considered when choosing an architectural solution as the chosen solution may have a large impact on network responsiveness.

Usability must also be considered. The solution must work on a variety of devices to ensure maximum user retention. This is another tricky requirement to meet, as it is difficult to tailor a front end that will work across the board.

In order to meet these functional requirements, I have assembled various technologies into categories and I will contrast them to ascertain the best fit for this project. I feel that the best way to do this is to lay out the technologies for the back-end, design a reference architecture and then decide on front-end technologies.

## Application Structure

#### Technologies

There are several ways that this system could be implemented. The two most likely are native applications and web applications. It would be possible to create the application for one specific native platform and then transpile it to the other, for example build the application in Android and then use a tool like Apache Cordova to port it to iOS. This does not allow for desktop applications however, so it would need to be ported to desktop also somehow.

The other alternative is to create a Web App. One of the newer versions of web apps is the Progressive Web App. This application will allow a web app that is run in the browser to work offline, and to be installable on the users device.

Whatever type of application will be used, there will need to be an underlying data structure. This will most likely be in the form of a database. The two major database players are SQL and NoSQL.

An SQL solution is older, more structured and less scalable than more modern NoSQL implementations. SQL is a better fit for more structured, normalised and relational data, whereas NoSQL is better for semi-structured/unstructured data that needs to be adaptable.

For the application layer, the two major technologies I am considering are Java (through the Spring framework) and NodeJS (through the express framework). Since this system will deal with large volumes of data, Java would have a performance edge in processing this data, but would be less efficient at handling mass concurrent requests. Node has the opposite problem, since it is asynchronous and single threaded, it can handle lots of mass requests, but is very slow at handling large data processing.

#### Firebase

I am also strongly considering Firebase integration into the project. Firebase is a solution that is generally used to provide server functionality to a client without the need for server-side code. It allows a client to read/write directly to a database, without server intervention. It also provides client-side file storage to allow users to save and retrieve files without the need for a separate server. Firebase also provides free hosting for static websites, served from a distributed Google infrastructure, served over SSL, with support for free SSL certs on custom domains too. (Google (A), 2017)([Google (A), 2017](https://github.com/JavaTheNutt/final\_year\_project\_documentation/blob/master/docs/design\_docs/README.md#google-a-2017))

##### Client-Side Database

Firebase provides two options for Real-Time data access, Real-Time Database and Cloud Firestore. Both provide integration with other Firebase modules, such as Authentication so that other processes managed by Firebase can be seamlessly integrated into operations involving the database.

###### Real-Time Database

This is the original Firebase Database offering. It is a simple key value store which allows data to be structured in a tree fashion. There is no true querying, instead event listeners are attached to a particular key, and a deep watch is maintained on the data stored below that tree. (Google (E), 2017)([Google (E), 2017](https://github.com/JavaTheNutt/final\_year\_project\_documentation/blob/master/docs/design\_docs/README.md#google-e-2017)) This is a very attractive method for storing hard and fast values that are often changed externally. It would also be very useful when dealing with a component based client, as a listener attached to a small section of data could be attached to a component, and only be active while the component is active.

###### Cloud Firestore

Cloud Firestore is the newest data solution offered by Firebase (04/11/17 its still in Beta). It acts more as a collection based database (structured like MongoDB) where the client app can store *documents* in *collections.* From the documentation it appears that this database would act like a Mongo database wrapped in a GraphQL server. So the server is specifically set up to perform deep queries on JSON collections. This provides a more robust method of querying, but at the expense of not being able to add listeners to a specific data set. (Google (F), 2017)([Google (F), 2017](https://github.com/JavaTheNutt/final\_year\_project\_documentation/blob/master/docs/design\_docs/README.md#google-f-2017))

###### Chosen Solution

I decided to use the older offering of Firebase Real-Time Database over Cloud Firestore. The reason for this is that I felt that Cloud Firestore was a more complete solution for web apps that did not have a server to support them. I felt that since this product was still in beta, it may not be the best choice to depend the entire backend architecture on.

Since Firestore appears to act as a REST API (or, rather a GraphQL server), I felt it would be a safer choice, as it would be easier to migrate from a custom API to a total Firebase data storage solution.

##### Cloud Functions

Firebase also provides a service called Cloud Functions. These are small NodeJS scripts that can be attached to events within the Firebase ecosystem. For instance, when a user signs up, a custom function can be run, perhaps sending a verification email to the user. There are a large selection of events that can be listened for, such as writes to a specific database node, or a user asking to reset their password. (Google (D), 2017)([Google (D), 2017](https://github.com/JavaTheNutt/final\_year\_project\_documentation/blob/master/docs/design\_docs/README.md#google-d-2017))

##### Authentication

The main attraction of Firebase is its authentication module. Firebase provides both client and server API’s for authentication. This method of authentication would allow the client app to authenticate with Firebase using an email/password combo, or a third party Authentication provider. Firebase then handles all local authentication through a combination of browser cookies and WebSocket connections. When the client needs to make a request to the server, it can fetch a JSON Web Token (JWT) from Firebase to be attached to the request headers. (Google (B), 2017) ([Google (B), 2017](https://github.com/JavaTheNutt/final\_year\_project\_documentation/blob/master/docs/design\_docs/README.md#google-b-2017))

On the server, Firebase provide an API to validate and edit the claims attached to the JWT. This will allow the server to tell Firebase what data should be included in the next token that is issued to the client. The server can then verify that the request can from a valid user, and which user made the request. (Google (C), 2017)([Google (C), 2017](https://github.com/JavaTheNutt/final\_year\_project\_documentation/blob/master/docs/design\_docs/README.md#google-c-2017))

This means that on the client side, I can abstract the details 0f dealing with various Authentication API’s (such as different OAuth Bearer Tokens from different API’s) into a single API which provides some powerful functionality on both the client-side(such as the wrapper for storing an authentication cookie to allow users to remain logged in between browser sessions) and server-side(such as the ability to add custom information to the Bearer Tokens to prevent unnecessary database reads on every request) API’s.

#### Structure

Figure 3--Full Physical Implementation shows my eventual chosen architecture. This architecture combines the best of both worlds from both the application layer and the database layer. On the application layer, Node acts as a gateway service and REST API, while a Java Application Server handles large data processing. This method reduces the amount of processing on the Node Server Processes, while also reducing the amount of strain from concurrent requests from clients on the Java Server.

Firebase is used to offload critical processes so that my servers will not need to handle them. These services include authentication, file storage and a client database that can be consumed using Web Sockets.

Google Cloud Platform API’s can be used for many tasks. In this project the one that will primarily be used will be the Maps API to allow geocoding of memonic addresses into geospatial coordinates. There are a variety of other tools available on this platform, such as Business Intelligence API’s and AI tools for chatbots which will need to be investigated further to see if they can add value to the project.

There will be both a SQL and NoSQL databases in the landscape. The idea behind this is that the Mongo database will store data directly pertaining to the users and the SQL database will be used for more structured, anonymised data that can be used for larger queries and aggregations.

In this scenario, the NodeJS REST API will act as a gateway and will return mainly the data stored in the MongoDB. For larger requests that need heavy processing, the Node REST API can make requests to the Java AS to offload this CPU intensive processing. The Java AS will also perform aggregations on the data and save aggregated results in the Mongo Database for the Node API to serve to the client.

There are also two Web Worker Processes in this landscape which will allow data transformation between the Firebase Database and the Mongo Database, as well as between the Mongo Database and the SQL Database.

To scale this down, the SQL database can be removed, along with the associated Web Worker processes. The Java AS can also be removed and associated processing moved to the Node Server Process for the time being. This architecture would look like Figure 4--Mid Physical Implementation.

Then, in order to scale this down yet again, the web worker process to transform data could also be pushed to the Node REST API, to remove that system from the landscape. The Mongo sharded cluster could be reduced to a single Mongo instance. The same could happen with the NodeJS server processes, which would mean that Load Balancer could also be removed.

This architecture could look like this what can be seen in Figure 5--Small Physical Implementation

In this case, there is just one Node process, one Mongo instance, Firebase, GCP and the client in the landscape. This reduces the scope of the project considerably and helps to make it easier to create a MVP, while also allowing a solid Architecture Transformation Plan to help reach the desired architecture eventually.

Building it in this method, with the Node Server Process first, rather than the Java AS, means that the endpoints that the client consumes data from will remain unchanged as the architecture develops. With this method, all of the systems that provide data to the client will be in the landscape from the beginning, providing peak user experience from the beginning.

The client does not really need to be set in stone at this point. With the architecture described above, the client could be either a native application or a web application, and the choice of framework to create this will have little or no structural impact on the landscape.

#### Infrastructure

The main infrastructure considerations will be for the Application Server. In this case there are four prominent providers that I am considering. These are Microsoft Azure, Google Cloud Platform, Amazon Web Services and Heroku.

I am currently thinking of deploying the first prototype on Heroku as it provides the simplest way of hosting an application server, and then migrating to either AWS or GCP for the production version, as they provide a much more performant solution.

For the front-end it matters less, since it will simply be a collection of static files that need to be hosted. My most likely solution for this will be Firebase, which provides static site hosting from Googles edge servers as well as a free SSL cert.

For the database, I am considering hosting the prototype on mLab for simplicity and then migrating to Mongo Atlas for the production version. Mongo Atlas provides far better performance, but requires more configuration to set up.

# Appendix

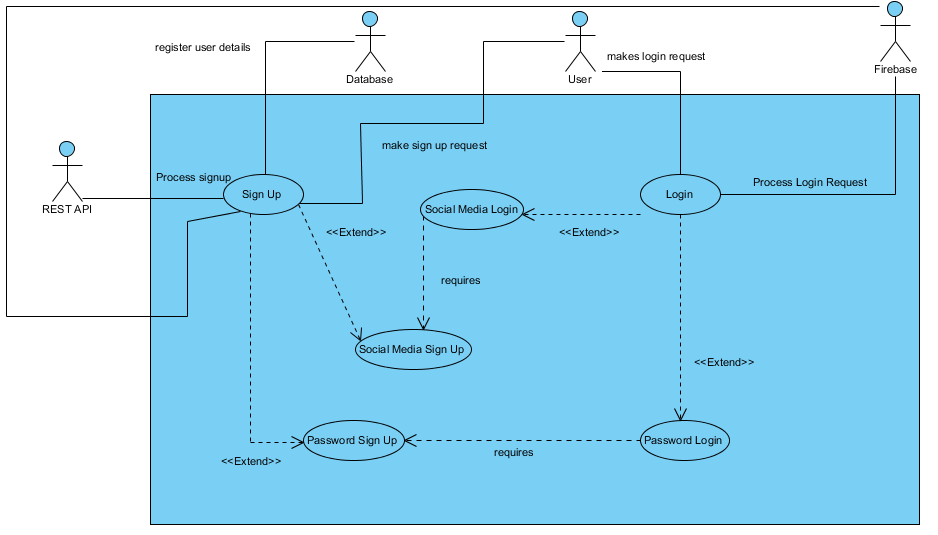


Figure 1--Auth Use Case

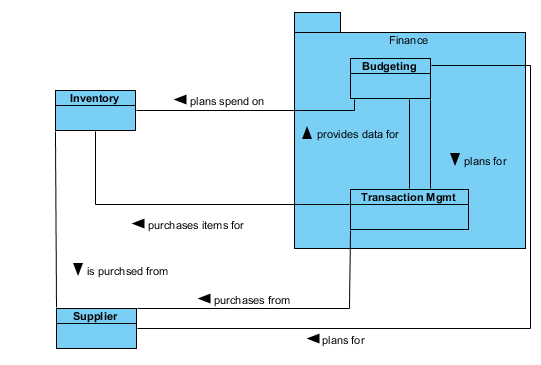


Figure 2--Conceptual Module Overview

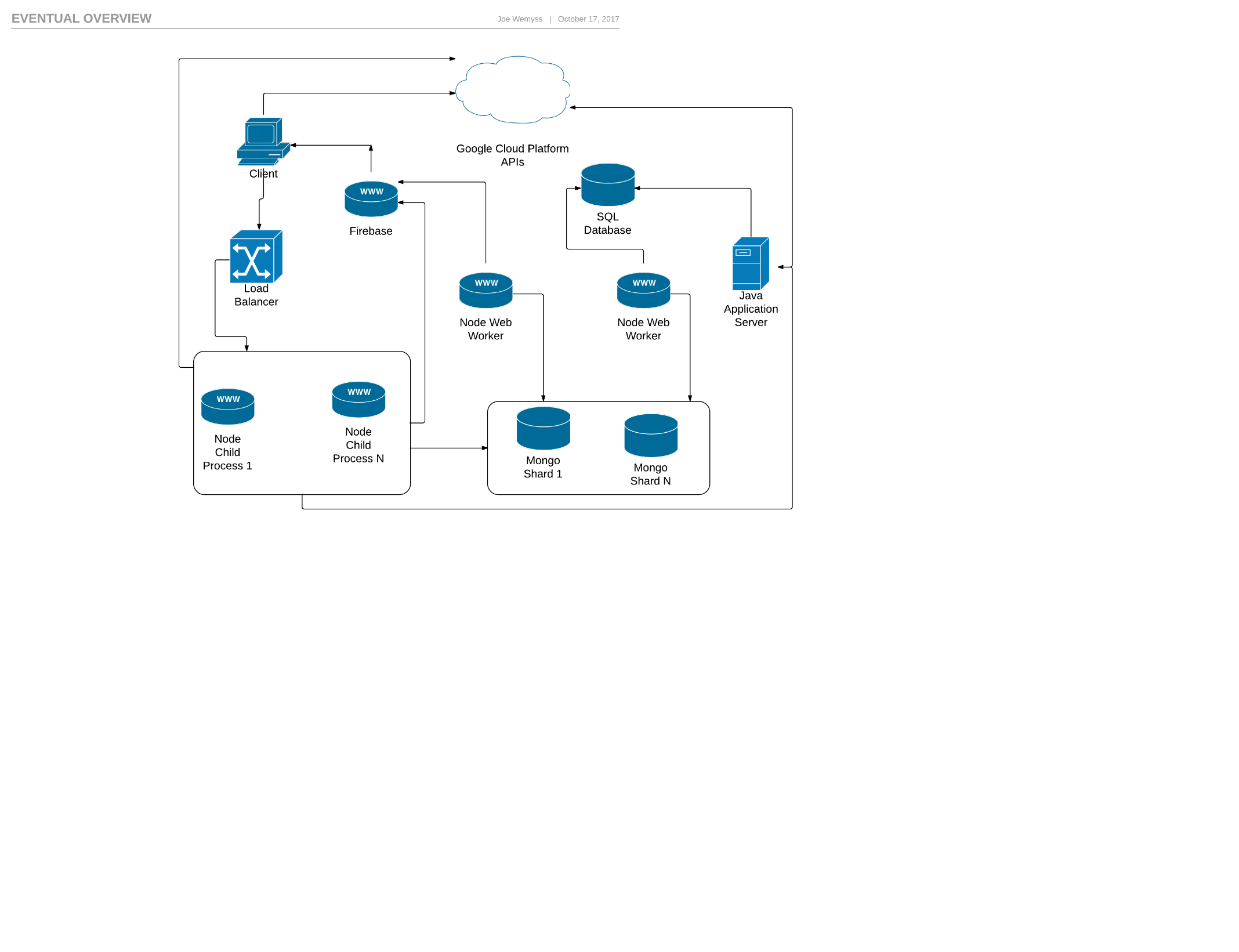


Figure 3--Full Physical Implementation

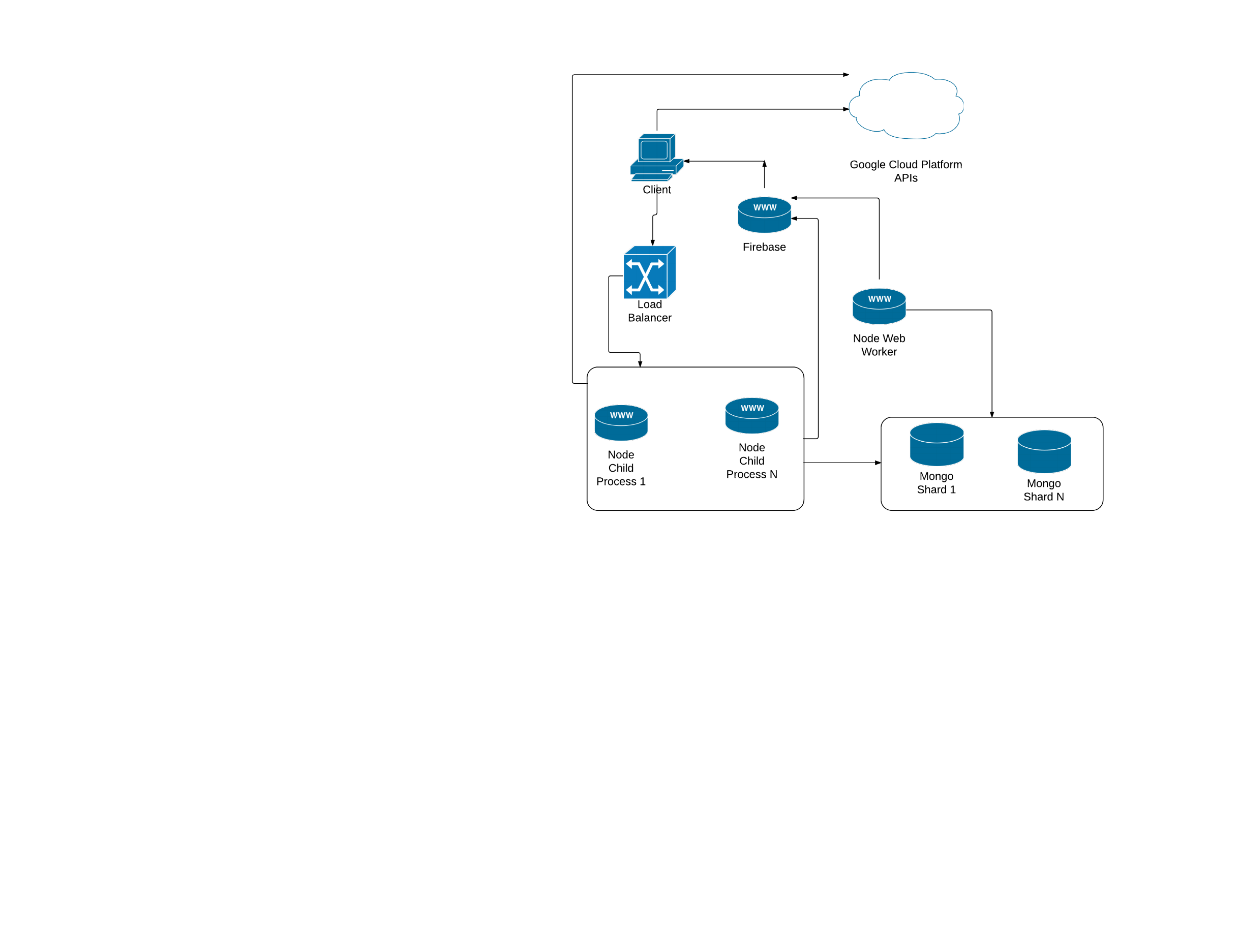


Figure 4--Mid Physical Implementation

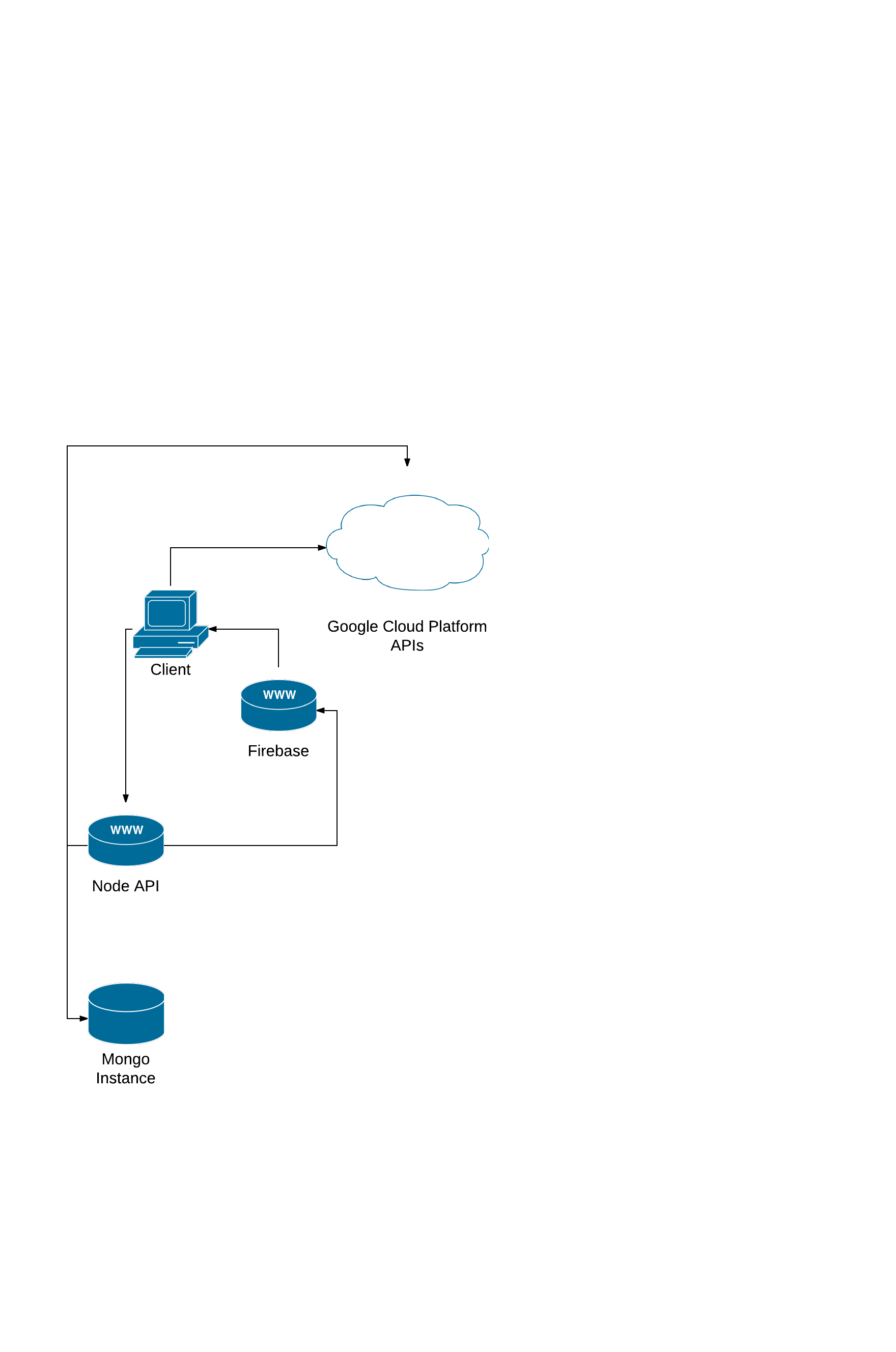


Figure 5--Small Physical Implementation

# References

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].

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].

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

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

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

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

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

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

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].

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