Dublin Institute of Technology
Department of Computing

Mobile Backend as a Service iOS

Timothy Barnard
Student number: C13720705

Supervisor: Edina Hatunic Webster
Second Reader: Dr. Susan McKeever

Submitted in part fulfilment of the requirements for the degree of
Computer Science, 4th April 2017

Abstract

Mobile applications are what we use in our everyday life, and over the past several years where mobile application development has expanded, so has the number of new developers wanting to build applications. This project will concentrate on iOS mobile development with the new programming language Swift. The number of YouTube channels have surged to meet the demand of iOS development education, but there still can be room for improvement. Some new and experienced developers can be put off designing and creating new applications due to time and resources. Cloud services that govern the mobile industry are not cheap and easy to use, and this can be an obstacle for making mobile applications.

The aim of this project is help speed up development and testing of mobile apps, to be able to publish quicker. This will give a service that does not cost anything, and that the developer has full control. The idea behind this will not only help in the development and testing sections but also when the application is published. To be able to give the app a new look and feel without requiring a complete build and publishing, and to allow the end-user more freedom with the design. This build will do alongside Apples strict and fair guidelines to ensure its validity.

Acknowledgements

I would like to express my gratitude towards my supervisor, Edina Hatunic Webster for her continued help. Her guidance to keep me on tract for the deadline, and giving me continuous feedback.

I would also like to thank Susan McKeever for being my second reader, and for helping with getting outsourced developers to give me feedback on the project. This has helped me further with not only getting constructive feedback but also validating the project.

The project would not be completed on time without the help of two mentors, and been a pleasure working along both of them.

I would like finally thank my family and friends for being there thought my college years.

Declaration

Declaration

I hereby declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed:

_____

Timothy Barnard 4th April 2017

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

1.1 Overview

The project title contains two parts: a backend, and a service. The backend itself consists of three parts: a server, application, and a database. When booking a flight, the website you open known as the frontend is where the user enters their information. These customers information needs to stored in a single location, so when the customer signs back in, their information still exists. This information is stored on a server in a database, known as a backend.

A service provides tools for mobile apps to interact with the backend. It is a way for developers to link their applications to backend cloud-based storage and services. A backend as a service or BaaS is best described by a tech analyst who refers to it as turn-on infrastructure for mobile and web apps. [5].

1.2 Project Objectives

The aim of this project is to develop a template for the development of mobile applications. This template is to help improve the development of modern mobile applications for new and experienced developers. The mobile backend as a service provides a selection of tools and services that enable rapid development of sophisticated mobile solutions. These services provided are given to help with the different phases of development. The project aim is to create a complete package that will enable new and experienced developers to speed up in the following areas:

1. Development

2. Testing
3. Production

The project deliverables to aid with the three phases include the following:

1. Mobile Back-end as a Service

   This is a model for developers to link their applications to the backend cloud storage and application programming interfaces (API's) exposed to provide the communication with the list of services above.

2. Dashboard

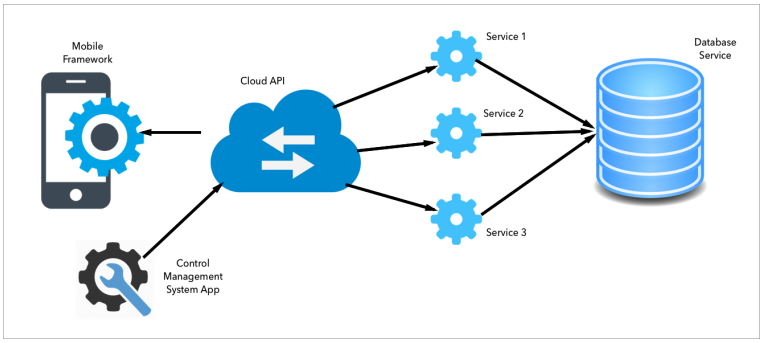
   The dashboard is a control panel that simplifies configuring the web server. It also contains data visualization tools to monitor the mobile app's activity.

3. iOS Framework

   The services above need a way to communicate from back-end to the mobile apps. This is accomplish by providing a custom software development kits (SDKs).

1.3 System Overview

Figure 1.1: Project Overview 2

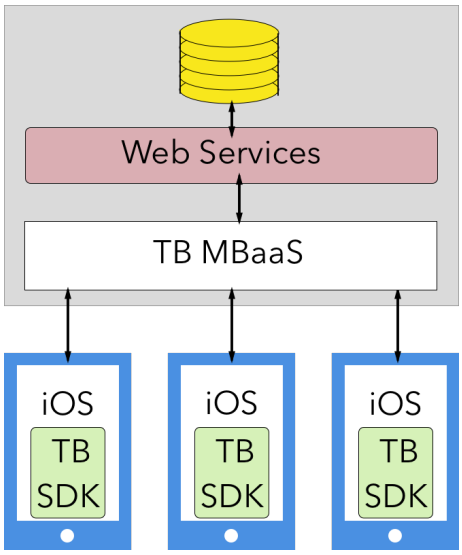


The two figures 1.1, 1.2 illustrates the overall project overview. The figure 1.1 shows the three deliverables and how they are connected to each other. In figure 1.2 contains iOS devices in which the SDK framework will reside. The backend containing the web sever with services and the database.

1.4 Project Challenges

The key challenge of the project was to find what developers functional requirements looking for when choosing a mobile backend as a service what current providers do not offer. This required setting up meetings with professional mobile developers and getting their opinion on the project.

Figure 1.2: Project Overview 2



The big challenge faced is to find a way to enhance the developing stage of any application, giving more power to the developer when the application has been published. This lead to another challenge to see how far Apple would allow applications to be configured after published.

1.5 Chapter Walk-through

Chapter 2 Research This chapters discusses the alternative existing solutions to the problem. It also goes into the potential technologies that could be used in the project along with the resultant findings. The technologies discussed will be what web server to use, the programming language and what the dashboard be development in. This chapter will include surveys and out-sourced discussions with professional mobile developers.

Chapter 5 Architecture The architecture chapter will show the overview diagram of the complete project, along with diagrams of the individual services.

Chapter 4 Design This chapters goes into the design of the projects. What methodologies will be used?, and go into detail of the list of features already mentioned in project objectives section.

Chapter 5 Development This chapter will discuses the different deliverable in the project in order to achieve the project aim. It will contain detail description of the features that will be used to aid in mobile application development.

Chapter 6 Implementation Implementation chapters explains how to use different deliverables included in the project. Along with code snippets which show how to set up the system and use the SDK.

Chapter 7 Testing/Evaluation This chapter will give an overview of all testing carried out including test-driven approach and unit service testing. The evaluation section discusses the review given from the out-source professional developers. The review not only includes their feedback using the system but also their recommendations where to from here.

Chapter 8 Conclusion The conclusion chapter contains a summary of the project overall and ends with a reflection of the project.

Chapter 9 Appendices The conclusion chapter discusses the meetings and evaluations with out-sourced professional mobile app developers.

Chapter 2

Research

2.1 Introduction

The primary focus of the project is to provide new and experienced mobile developers with three deliverables. The deliverables will bring something new to the table, a new way of developing applications. These deliverables are as follows:

- Mobile Backend as a Service
- Mobile framework (SDK)
- Dashboard

This chapter will cover not only the background research required for the deliverables but also research from mobile developers and general mobile users. It will also include the technologies required along with the current similar solutions.

2.2 Background

2.2.1 Mobile Backend as a Service

BaaS is an approach for providing mobile app developers a way to connect their application to back-end cloud storage and processing while also providing standard features such as user management, push notifications, storage, and other features that mobile users demand from their apps these days. The objective of any developer is to get their product finished and publish as quickly as possible. By providing an MBaaS to developers, it

reduces time and resources needed to develop an app. As a research paper from Kinvey [5] states these points regarding what BaaS delivers:

- Efficiency Gains - Reducing overhead in all aspects of mobile app development, increasing efficiency at all stages of development
- Faster Times to Market - Reducing the obstacles to take a mobile app from idea to production and overhead with operations once in production
- App Delivery With Fewer Resources - BaaS supports development with fewer developers and supporting data and IT resources
- Optimize for Mobile and Tablets - BaaS providers have put a lot of time and resources into optimization of data and network for mobile apps, and reduce fragmentation problems across multiple platforms and devices.
- Secure and Scalable Infrastructure - BaaS provides a bundled infrastructure that deals with scalability, security, performance and other operational headaches, leaving developers to do what they do best
- Stack of Common API resources - BaaS brings common and essential 3rd party API resources into a single stack, preventing developers from having to go gather them separately

After reviewing why developers should use an MBaaS, the research paper goes on to discuss a pattern of building blocks being to emerge. These are the essential services that any MBaaS provider should be offering.

- User Management - It all starts with a user, the ability for users to sign up and log in.
- Storage - a central location for all app data to be stored to connect all users together.
- Rest API - the link between the back-end services and client applications.
- Communication - feature such as push notifications to keep users connected live with other users.

2.2.2 Mobile developers

The fundamental part of this project is to research from experienced developers, asking them in person and forums. A list of questions was put up on different forums regarding the functionality required for the mobile backend as service. These include: What advantages and disadvantages do you find with current third party mobile back-end as services?, What features do you want to be included that are not currently available?, Where do you see the future of third party services going, are they required or should mobile developers implement

their own back- end?. This, however, was not successful, as comments were coming backing saying that the questions should be problem specific.

The next plan after the forums did not help, was to meet with Trust5 [6] and Tapadoo [7], two mobile software developing companies based in Dublin City to discuss my project proposal. The general feedback was positive, seeing the potential and powerful tool it could be. A working prototype iPad and iPhone app was used to demonstrate the remote configuration service as part of the project which updates a client mobile app interface which will explain later.

This enabled them to get a feel for the project, and both companies gave constructive feed; where to be careful and keep parts to look out for. Tapadoo developer discussed that the A/B testing is a plausible key feature to use along with the remote configuration, he also mentioned to keep close to Apple guidelines. Trust5 developers mentioned the project can give developers the capability of creating a white label product, that one app can be used for different products.

See appendices chapter for the complete case study of each company.

2.2.3 Mobile users

Surveys were created to help get mobile users feedback using apps. The reason for this to be the part of the research is that they are the end-user. They are the ones we want to keep happy, keep using our apps. The survey contained some questions which along with the results are as follows:

1. Would you delete an app if it crashes? Figure 2.1

Figure 2.1: Question 1

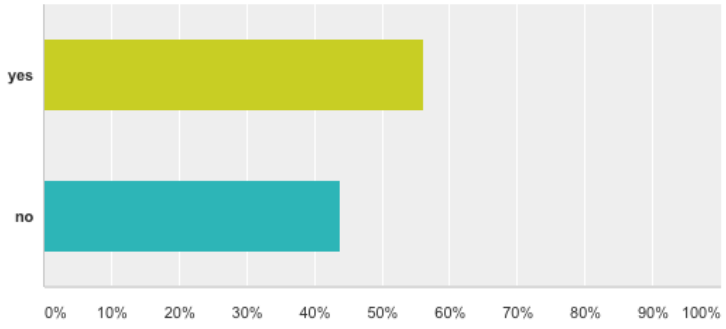


Figure 2.2: Question 2

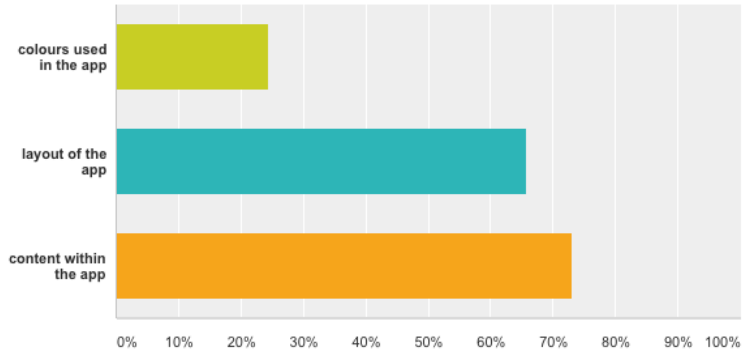
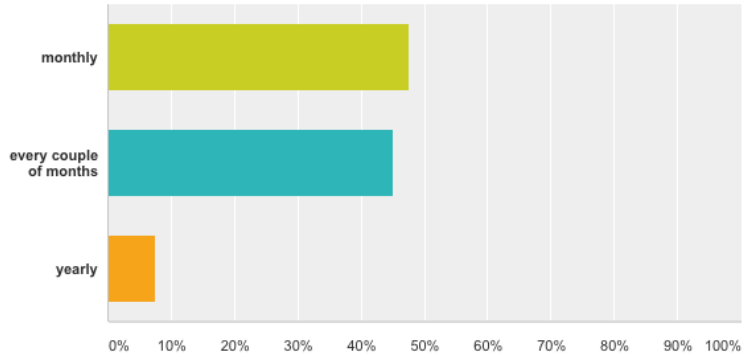


Figure 2.3: Question 3



2. If you had a choice between two of the same type apps, Do you choose based on? Figure 2.2

3. How often do you like app updates? Figure 2.3

4. Would you prefer the user interface to be kept up to date and fresh more often? Figure 2.4

Figure 2.4: Question 4

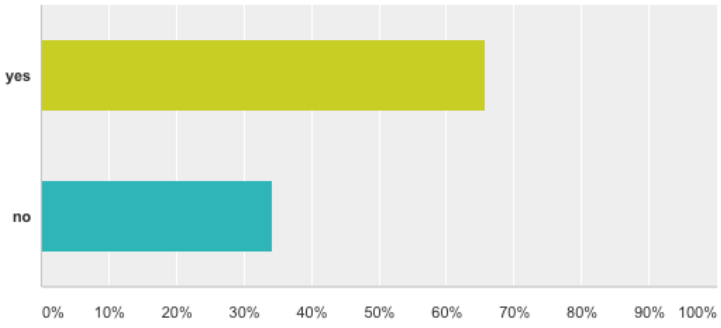
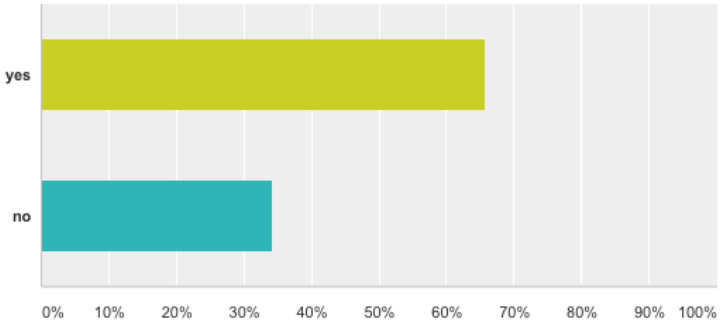


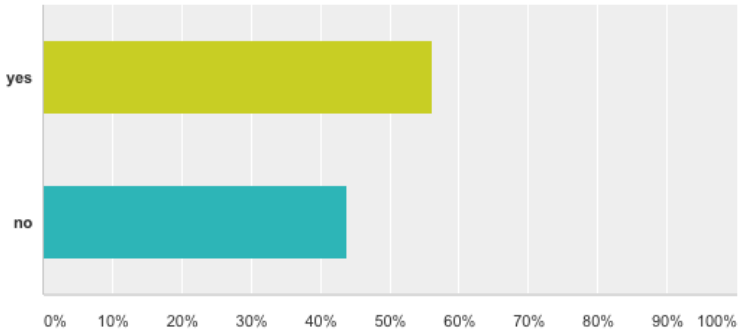
Figure 2.5: Question 5



5. Would you prefer being given the option for language choice (e.g. English)? Figure 2.5

6. Do you mind apps collecting analytics in the background to help improve the app ? (not personal information) Figure 2.6

Figure 2.6: Question 6



The above survey gives an understanding of what users feel when using apps, also some insight into what they want to have more often. By asking users these questions, it has given a significant reason to implement one of the key features to be implemented in the project. A feature that enables developers to keep the app regularly updated for users to feel like they are not forgotten, to give the app a new look and feel. But also a quick way to stop the user from deleting the app because it crashes. This feature will be further discussed in the design chapter.

2.3 Technologies

2.3.1 Programming Languages

Objective-C

Objective-C [8] is an object-oriented programming language developed in the early 1980s. It was used to develop on NeXT OS which later became OSX and iOS. Objective-C is a superset of the C programming language meaning that it is possible to compile any C program with an Objective-C compiler.

Swift

Swift [9] is Apples latest open-source programming language used mainly for developing on iOS, macOS, watchOS and tvOS applications. Swift it an alternative to the Objective-C language, but is sometimes referred to Objective-C without the C. In contrast to Objective-C, it does not expose pointers to refer to object instances. Swift does retain Objective-C concepts including protocols, closures, and enums. It was designed to make writing and maintaining programs easier for developers.

Another big difference between Objective-C and Swift is Protocol Oriented Programming (POP) in contrasts to Object Oriented Programming. Swift is the first language to be POP. Protocols do not provide any implementation, the only provide the blueprint of methods and properties, things that you need to accomplish when writing code. It describes what must be implemented, and what it will look like. They then can be adopted by a class, struct or enums, and then those classes, structs will provide the implementation required.

Swift also provides protocol extensions, which has an implementation of the protocol methods. This can be used to reduce code by where a list of objects implementing the same functionality, these then can be placed inside a protocol extension. Thereby following the principle of Dont repeat yourself. [10] An added benefit to the protocol extensions, is that they can be used to extended an already built-in a structure such as a dictionary with extra functionality. Thereby not needing to refactor to implement the new dictionary class.

Python

Python [11] is a high-level programming language. It is designed to allow programmers to express concepts in fewer lines of codes than possible such as Java or C++. It is the perfect language to write programs on both a small and large scale. It supports object-oriented (OOP) and functional programming. Python was first created in the late 1980s by Guido van Rossum as a successor to the ABC language.

2.3.2 Web Frameworks

Perfect (Server-side swift)

Perfect [12] is a web server and toolkit for developers using the Apple's Swift programming language to build applications and other REST services. The web server gives the ability to deploy on both the macOS and Linux servers. It provides Swift developers with the tools to develop a lightweight, maintainable apps. It is also a free open sourced software with good community support on Slack and Twitter applications.

Kitura (Server-side swift)

Kitura [13] is a web server and web framework using Swift 3 developed by IBM. It is similar to Perfect, using core Swift technologies, and a light-weight web framework. It also is a free open sourced software with abundant of community support and documentation.

Django

Django [14] is an open-source high-level Python web framework which follows the Model-View-Controller (MVC) pattern. It consists of an object-relational mapper (ORM) that maps models (M) defined as Python classes to a relational database, a system for processing HTTP request to a web view (V). It focuses on rapid development and the principle of don't repeat yourself. It was created in 2009 by few web developers and began to use Python to build applications. Django looks after authenticating the user when signing up, signing in and signing out. A popular site such as Pinterest, Instagram, and Bitbucket use Django for their web framework.

Flask

Flask [15] is a micro web framework written in Python and initially released in 2010. Applications such as Pinterest and LinkedIn use this framework. It is a micro framework as it does not require any dependencies to run, as well as that it does not have extra layers such as database but supports extensions that can be added to create additional features. Flask is popular among Python enthusiasts and was the most popular Python web framework on Github.

Node.JS

Node.JS [16] is an open-source, cross-platform environment originally released in 2009. It is used to create a variety of tools and applications such as GoDaddy, Groupon, and Paypal. It is driven by events such as

when a consumer purchases an item. Node.js has been optimised for web applications with many input/output operations, as well as real-time communication.

2.3.3 Dependency Managers

CocoaPods

CocoaPods [17] is the de facto standard of package management for iOS. It has the largest community and is officially supported by almost every open-sourced iOS library. It contains over twenty-eight thousand libraries which are used in over 1.8 million apps. Cocoapods manages the libraries required for the mobile app in a single file called Podfile. This file not only contains the library name but also what version to use.

Carthage

Carthage [18] is intended to be the simplest way of add frameworks to apps. It was the first dependency manager for macOS and iOS, created by a group of developers from Github. It was also the first dependency manager to work with Swift. It exclusively uses dynamic frameworks instead of static libraries; this is the only way to distribute Swift binaries that are supported by iOS 8 and up.

2.3.4 Databases

MySQL

MySQL [19] is free open-source relational database management system (RDBMS) initially released in 1995. Previously owned by a Swedish company MySQL AB and now owned by Oracle Corporation. MySQL works on many systems such as macOS, Windows, Linux, FreeBSD so making it a common choice and reviews are positive.

MongoDB

MongoDB [20] is also a free and open-source database type program but is a document-oriented. Classified as a not-only SQL (NoSQL) which uses JSON-like documents to store objects. Features include indexing, replication, load balancing and the list goes on. The main difference of MongoDB is that it is not a relational database. Objects that are related somewhat together are stored together in one file, improving retrieval of data.

PostgreSQL

PostgreSQL [21] is an object-relational database with additional object features. It differs itself support for highly essential and integral object-oriented and relational database functionality, such as complete support for reliable transactions. It also free and open-sourced, yet very powerful with the capabilities of storing procedures.

2.3.5 Web servers

Apache

Apache [22] was created by Robert McCool in 1995 and has been furthered developed by Apache Software Foundation since 1999. The Apache web server has been the most popular server since 1996. Administrators choose this for its flexibility, power and widespread support. The Apache server creates processes and threads to handle additional connections. The admin can configure the server to control the maximum number of allowable processes, which is depending on the amount of physical memory. Each connection gets it own new thread which handles on user request.

Nginx

Ngnix [23] was developed by Igor Sysoev in 2002, which answered to the problem that web-servers began managing ten thousand concurrent connection. It was initially released in 2004. Nginx has grown due to its light-weight resource utilisation and its ability to scale. Nginx works different to Apache, in that it does not create new processes for each web request. Instead, the admin configures how many worker processes to create. The rule of thumb is that one work process for each CPU. Each worker can handle thousands of requests.

2.3.6 Cloud Computer Services (IAAS)

Amazon Elastic Computer Cloud (EC2)

EC2 is a web service that provides secure, re-sizable compute capacity in the cloud. It is designed to make web-scale cloud computing easier for developers. [24] Amazon EC2 is widely used by companies such as Netflix, AirBnB, and Expedia. It can be used to launch as many or as few virtual servers as needed, that can be configured to suit your needs.

DigitalOcean server

DigitalOcean [25] is an American cloud infrastructure provider. It provides developers cloud services that help to deploy and scale applications that run simultaneously on multiple computers. As of December 2015, DigitalOcean was the second largest hosting company in the world regarding web-facing computers. Within a few steps, you can have the server up and running.

2.3.7 Evaluation

Table 2.1: Findings

Technology	Version	Area
Swift	3.0.1	Client Framework/Back-end Language
Linux	16.0.4 X64	Server OS
MongoDB	3.2.11	Cloud Storage
Flask	0.1.1	Test Web Framework
CocoaPods	N/A	Dependency Manager
DigitalOcean	N/A	Cloud Service
Nginx	1.10.3	Web server
Perfect	2.0	Web Framework

Programming Language

Chosen technologies as a result of the research for client-side application is Swift for both the framework and dashboard app. I decided to go with Swift for client-side programming for few different reasons, but one that stood out was a blog from 9To5Mac website [26]. At the beginning of 2016 Swift took over Objective-C making it the 14th most popular programming language. Protocol Oriented Programming (POP) is an interesting concept mainly used in Swift. It is starting to become commonly used instead of Object Oriented Programming (OOP) because objects are things that encapsulate complexity. So an object does it in a certain way, but protocols can provide objects with doing it different ways. With POP, the interface being what the user interacts with it the primary and only concern.

Dependency Managers

A-Coding blog review goes into the pros and cons of both CocoaPods and Carthage [27]. CocoaPods has a large community behind it so that problems can easily help and resolved. There are a large number of available libraries and are listed on their website: `https://cocoapods.org`. It has a centralised file that manages all libraries required for the app along with version. But because it is centralised, if it goes down you will be affected.

Carthage is not centralised; each dependency is fetched from their original repositories. It can be very slow managing all dependencies, especially if the app requires a large number of them. Some find CocoaPods invasive as it modifies the Xcode project by building its workspace. However, Carthage does not; it is up to the developer to add the required libraries into their project.

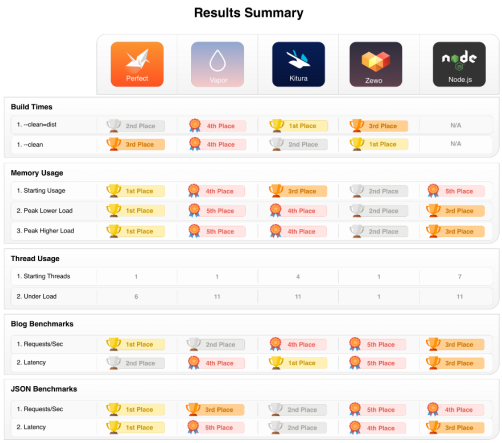
Due to extensive help from the community, and the project is all about helping the developer with reducing the amount of development, the project will be using the CocoaPods dependency manager for the MBaaS framework.

Web Framework

The project is aiming to be open-source so that developers can contribute towards it. So without needing to learn a different language like Python for Django, the choice is a web-server using Swift language. As Swift is becoming more popular and has become open-source, it is reasonable conclusion that the near future will include more back-end servers written in swift.

Now the web framework chooses it based on Swift programming language; the choice is between Perfect and Kitura. A benchmark was conducted by Ryan Collins, a web developer. He posted a blog about the outcome [28] which clearly shows that not only is both Perfect and Kitura faster than Node.JS but that Perfect is the leader. The blog goes into detail of the test that was run, the server setup and the number of requests made.

Figure 2.7: Benchmark



The result as stated in the blog is that Swift is more capable of taking on the established server-side frameworks. So regarding what Swift web framework to use, the decision is Perfect with over 10,000 stars on the Github repository. [29]

Database

The database choice is MongoDB which is a NoSQL. It provides a dynamic way of storing data where object properties name and type are unknown until runtime. Using this database allows the system implemented with the capability of different kinds of data stored. From the article 'When to use MongoDB rather than MySQL' [30] the reasons for choosing MongoDB is when:

- Expect a high load amount of data
- Need to grow big
- Do not have a database admin

The above 3 reasons are compelling enough to choose MongoDB because when designing a back-end as a service for applications of any type, then freedom and flexibility are key. Another valid reason is when you do not know the database structure, and also when providing a service to developers to give them tools to create any database structure is another strong reason to go with MongoDB.

Cloud Computer Services (IAAS)

DigitalOcean is not an Amazon competitor; its market is completely different. DigitalOcean targeted audience is small developers who want to start up a small high-performance instance [31] quickly. This is the perfect reason why DigitalOcean is the chosen IAAS for this project; it is clean, easy to set up, and inexpensive deployment. Amazon, on the other hand, provides a wide variety of tools for different mobile services, but simplicity is key. That alone is the reason for choosing DigitalOcean.

2.3.8 Web servers

The two web servers Nginx and Apache are both close rivals, they both support multiple OS's, and provide larger community bases support. The are both free and open sourced, with good security and performance. In the end, Nginx was chosen down to the speed of request-processing and its light weight. [32]

2.3.9 Extra tools required

Xcode

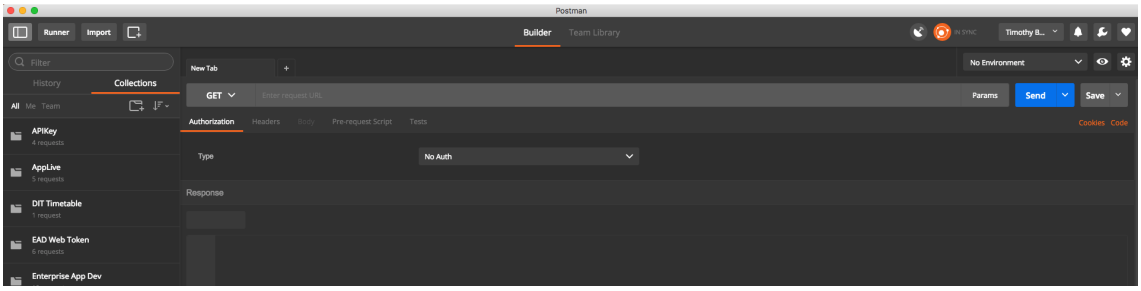
Xcode [33] is an integrated development environment (IDE) containing a suite of software development tools devel- oped by Apple for developing software for macOS, iOS, and tvOS. First released in 2003, the latest

version 8 is a free to download via Mac App Store. It supports source code for programming languages C, C++, Objective-C, and Swift, etc. This will be used in developing the dashboard and the web server.

Postman

Postman [34] is a powerful GUI platform that helps developing API faster and easier, from building API request to testing. The application can be downloaded on multiple of platforms as well as a chrome extension. In figure 2.8 illustrates the user interface.

Figure 2.8: Postman



Swift Playgrounds

Swift [33] playground also knows as a just playground is an interactive work environment that allows you see the values in the sidebar for the written code. As and when you make changes to your code the sidebar reflects the changed result. It was introduced in Xcode 6 and enhanced in Xcode 7 that make learning Swift and experimenting much easier. Instead of having to create an app just to run and test some Swift code, a smaller playground can be done to view the outcome of each test piece.

APNs Auth key

The Apple Push Notifications (APNs) [35] involves generating an authentication key, that will sit on the server and will send notifications to one or more devices. Until recently, the generating of the authentication key consisted of some painful steps. These include filling out a Certificate Signing Request in Keychain Access, then uploading it to your developer account. After which downloading a signed certificate, to which converting to .pem format. Also, the certificate would then expire, so the steps would need to be done again every year.

Now Apple has improved this service, which involves the creation of one key of type .p8, which does not expire. The downloaded file, can then without converting be upload to the server to be used. Apple also gives three text keys that are used in authenticating the APNs and send the notifications. These include; APNs Auth key, Team ID and the App ID.

Github

Github [36] is a web-based version control repository. It provides code sharing, publishing, deployment and much more. The developer can create as many private or public repositories as required. Github provides developers with tools to backup, version control and creates branches to separate out builds.

Perfect Assistant

"This macOS companion application is a set of convenience tools designed to help Server Side Swift developers start, manage, compile, test, and prepare for deployment more easily. From those expanding into backend Swift development for the first time to seasoned senior engineers working on enterprise level projects, the Perfect Assistant will facilitate your work." [37] It is used to help include all the required packages needed to build the project, and includes tools to deploy to Amazon web-server and docker.

OAuth 2

OAuth 2 [38] is an authorization framework that enables applications to obtain user accounts through services such as Facebook, Google, Github, etc. It works by delegating the authentication service to another third party which returns limited user information. This is a service that can be used to authenticate a user to access an application and is commonly used. OAuth 2 defines four roles:

- Resource Owner

  - the user who authorises an application to access their account

- Client

  - the application from which the user wants access

- Resource Server

  - where the users account is stored

- Authorisation Server

  - verifies the users identify then provides an access token

Table 2.2: Alternative Solutions

Service	Parse	Firebase	BassBox	Amazon
Notifications	Yes	Yes	Yes	Yes
Database	Yes	Yes	Yes	Yes
Analytics	Yes	Yes	No	Yes
self hosted	Yes	No	Yes	No
Remote configuration	No	Yes	No	No
Backup	No	No	No	No
User Sign-Up	Yes	Yes	Yes	Yes
A/B Testing	No	Yes	No	Yes

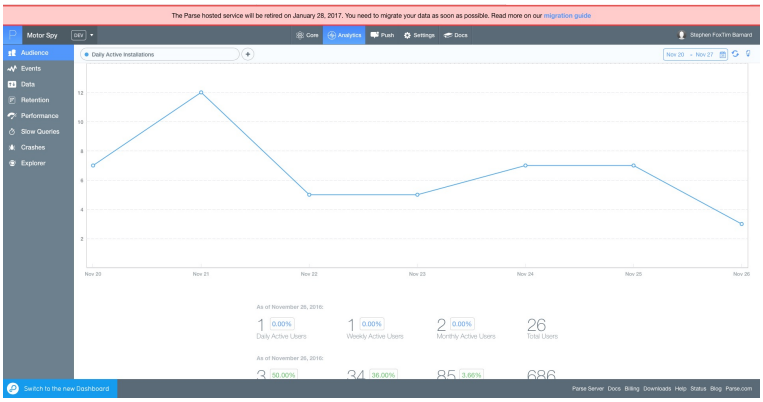
2.4 Similar Technologies

2.4.1 Overview

2.4.2 List

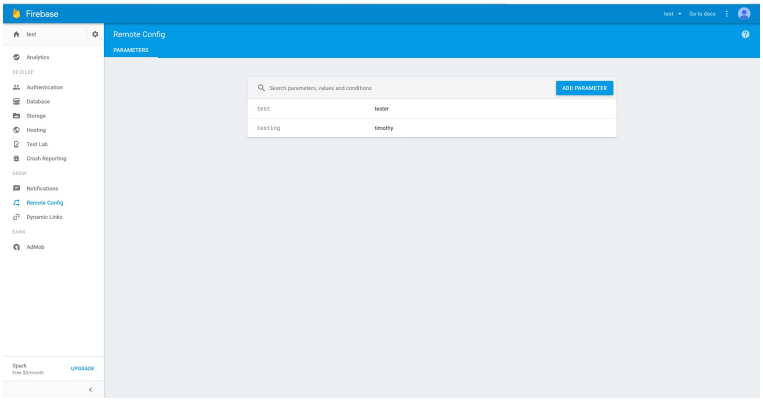
Parse

Figure 2.9: Parse



Parse [39] was founded in 2011 by Tikhon Bernstam, Ilya Sakhar, James Yu and Kevin Lacker former Google and Combinator employees. The project kick-started when it raised 5.5 billion dollars in funding in late 2011 and by 2012 over 20,000 mobile developers were using the service. Facebook in 2013 acquired the company for 85 million dollars and continued to grow. By 2014 500,000 apps were using the service, but sadly Facebook in 2016 announced that they are closing down the service in January 2017. They do provide tools and tutorials on how to migrate to your hosted service. The service when operational was widely used by developers and Fig 2.9 shows the main dashboard page. One of the developers mentioned in the interview that his company Tapadoo used to use Parse before they announced the closure of the service.

Figure 2.10: Firebase



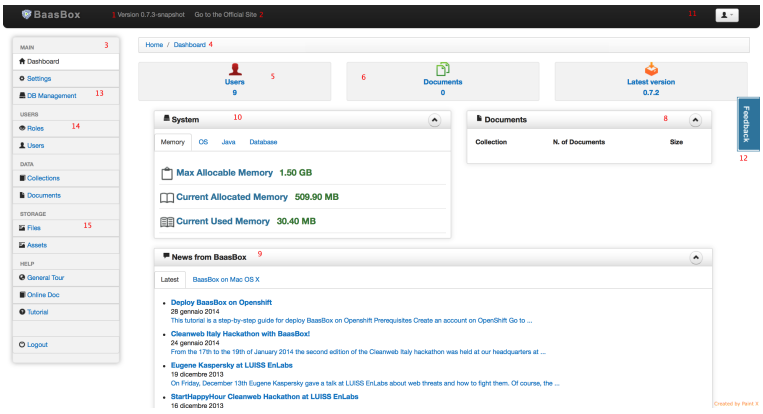
Firebase

Firebase [40] evolved from Envolve, a start-up founded by Tamplin and Lee in 2011. They provided a service that enabled developers to integrate on-line chat into their website but found after a while that the service was being used to pass application data that was not chat messages in real time. So the team decided to separate the chat system and real-time architecture that powered it. This lead to the founding of Firebase, a separate company.

It raised funding in 2013 and in 2014 Google acquired the company for an undisclosed amount. The company provides a list of services as shown in Table 2.2 for free for a limited amount of users and storage of 5GB, but as you increased the storage and users using your application then so does the price. If we way in more storage, real-time database space then the price per month is around 200 dollars [41]. The dashboard for managing your project showed in Fig 2.10; tthe current page is remote configuration service.

BaasBox

Figure 2.11: BaasBox

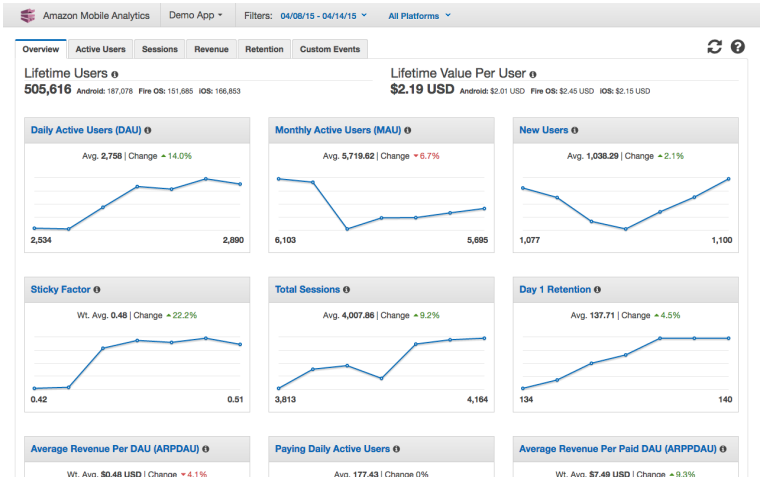


BaasBox [42] is an open sourced project founded in 2013; it is an application that acts a database and application server combined. It provides developers with an API as a backend to store data for their mobile and web

applications. It is the first back-end as a service to be open source and free to download. BaasBox also provides cloud services so instead of hosting the back-end on your server. Fig 2.11 shows the dashboard main page, on the left panel are a list of services they provide such as the database.

AWS

Figure 2.12: AWS



Amazon Web Services(AWS) [43] is a subsidiary of Amazon.com which offers a suite of cloud computing services launched in 2006. By 2007 Amazon claimed that more than 180,000 developers had signed up to use AWS Amazon Web Services. One of its services to AWS Mobile Services, a way to provide help to develop mobiles apps that can scale to hundreds of millions of users. Famous companies such as Airbnb and Netflix uses AWS mobile services to power their applications. Like other MBaaS providers, they do offer a limited 12-month free tier which includes 5GB of standard storage, 20,000 get requests, and 2,000 put requests. Then outside of these limits, the price does rise. The list of services they provide includes analytics shown in figure 2.12.

2.5 Prototyping

Dashboard

To aid in the project research, the development of an iPad app was required. This involved developing a initial application, which focuses on the key feature of this project called Remote Configuration. Remote Configuration will be explained in the design chapter, but briefly, it gives the developer the power to update the user interface objects of their applications without needing to publish a new version. This will be done using configurations files, which the mobile apps will download and read from.

This developed required three parts: the iPad dashboard app, demo mobile application, and the web server. An

already previously developed mobile application called DIT-Timetable was used, by adding the new functionality to update the UI object properties. Next, the server required services to send and retrieve the configuration files which will be used to update the apps UI.The DIT-Timetable app already has a backend, so adapting that to include the services made it easier. Lastly, the iPad was designed and developed to set these properties of the UI objects into JSON objects and update the server. The following figures show what the iPad app looked liked.

Figure 2.13: Main View

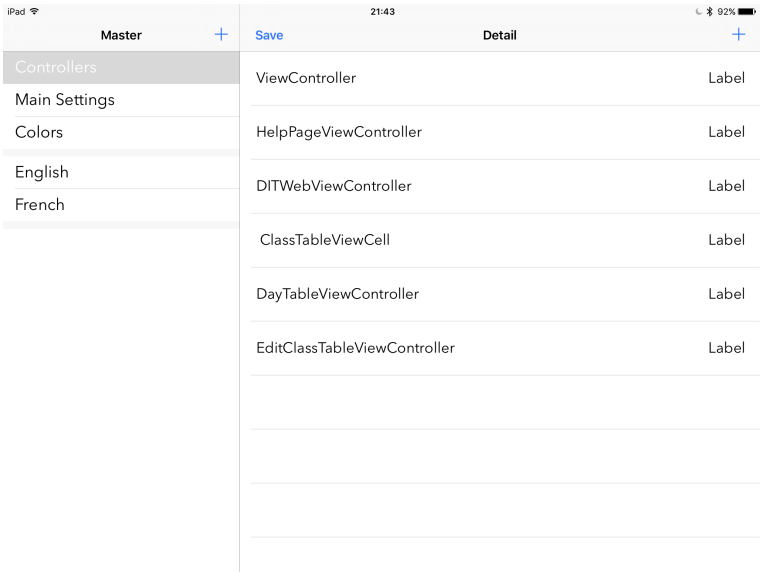


Figure 2.13 shows the main view of the application. The controller's tab holds the list of classes contained in the application. The reason for this design will be discussed in the design chapter later. Then each class will hold all the objects in each view, so when choosing the class, you are brought to the next screen seen in figure 2.14. But first, this view has the functionality to add a new class, along with saving this current configuration to the server, thus updating what the user sees in the DIT-Timetable app. In figure 2.14,once the users have chosen an object, the properties of an object can be updated. In this example, the background colour along with the text colour can be set. This brings us to the next figure 2.15 where the colours used can be updated. So for example, setting the background colour of the main view, or a label, etc.

Figure 2.14: Properties updating

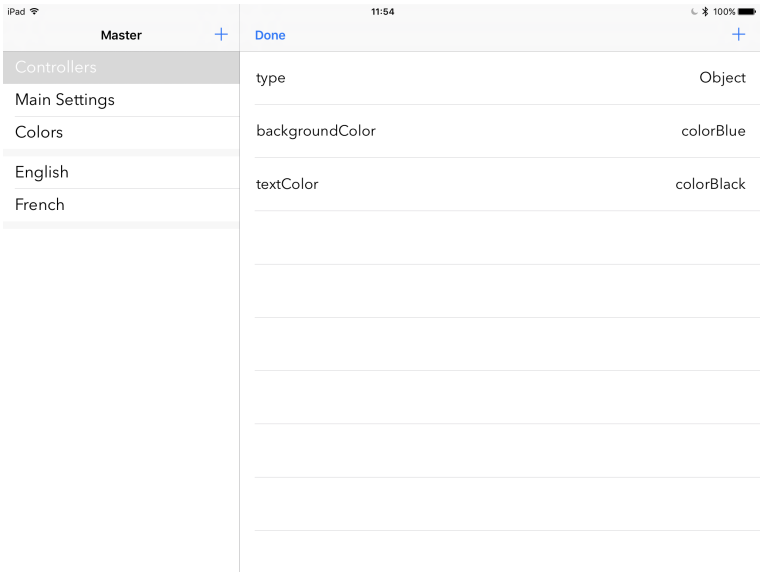
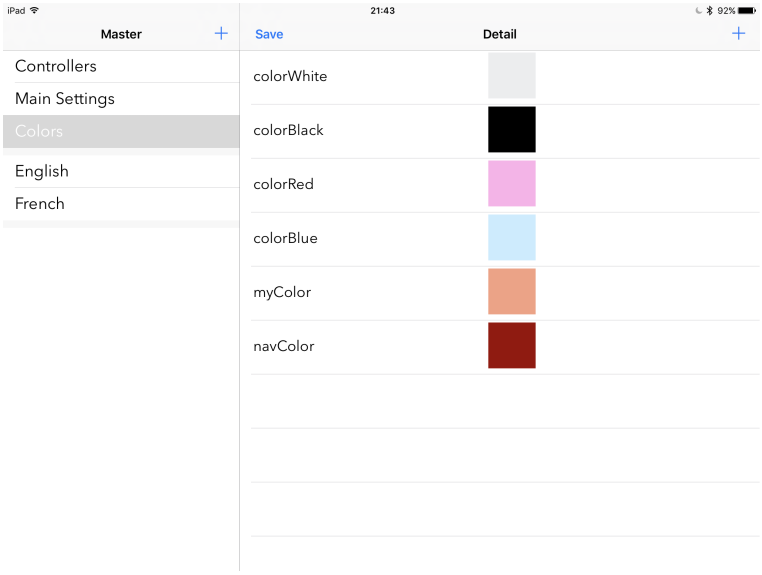


Figure 2.15: Properties updating



Chapter 3

Architecture

3.1 Overview

Figure 3.1: Overview

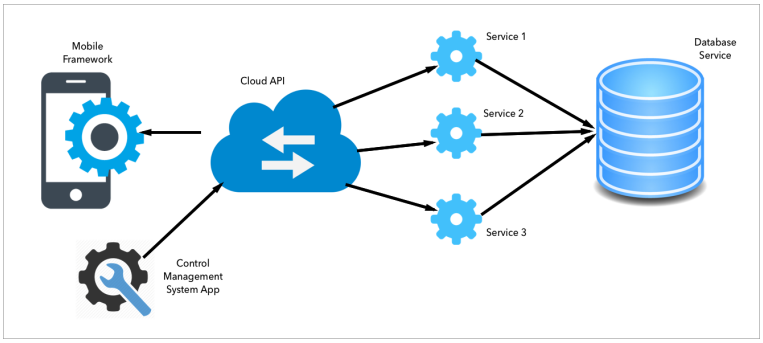
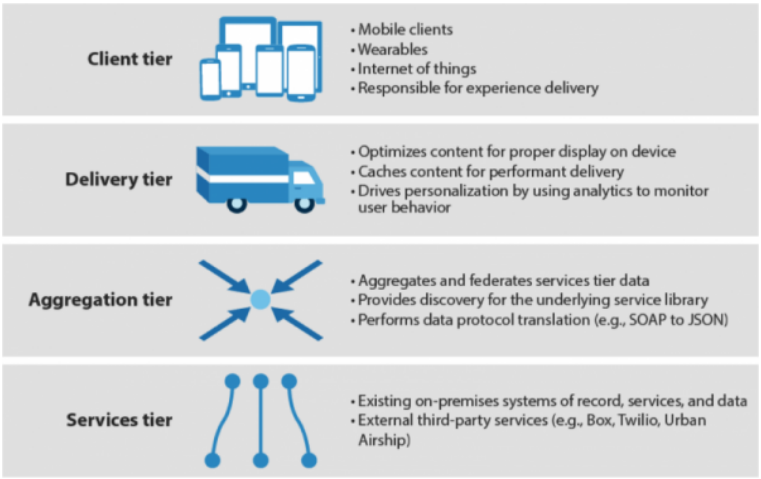


Figure 4.1 illustrates the overview architecture of systems and the four main parts. The first being the web-server that provides a cloud-based service, that takes requests and performs the necessary task. The next part which follows on from the web-server is the database, to store data persistently. This also provides a cloud-based database so that data can be accessed from anywhere. The third part being the mobile framework which creates the communication to the web server through an API which will be discussed later in this chapter. The framework separates the complexity of the web-server system and provides easy to use tools to communicate. The last main part is the control management system app, which provides an interface to configure the web-server and display the current set-up.

3.2 Web Architecture

[?]

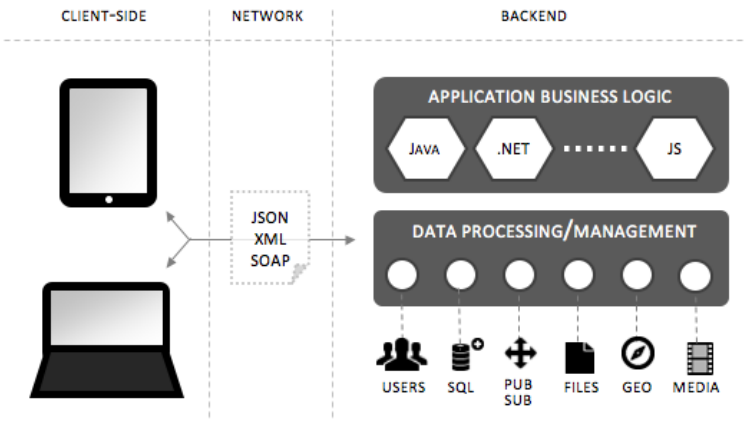
Figure 3.2: Four-tier Architecture [1]



The Four-Tier Engagement Platform is the chosen web architecture for my project shown in Fig 3.2. The client tier allows the development of an application without having to worry about the backend services. Delivery tier gives the consumer the best possible mobile experience by caching content locally on the device app, so if service is lost, then they can still use the app. The aggregation tier connects the apps to the right services with bi-directional, real-time data from the back-end. Finally, the service tier gives the other levels the data they require. It is also used to integrate current services already being used by the company such as MySQL or MongoDB.

3.3 Back-end as a Service

Figure 3.3: Client Server Diagram [2]



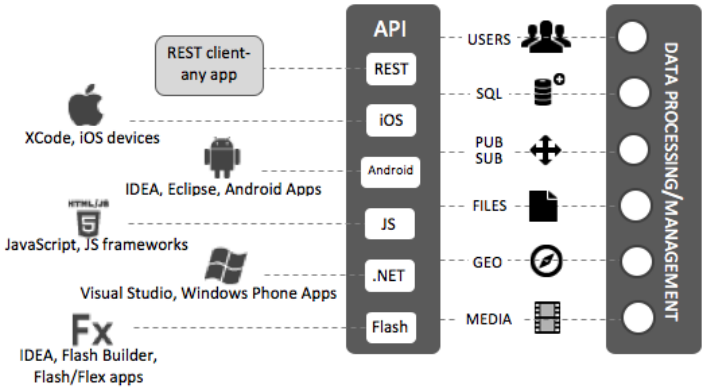
Applications today are put into the category of client-server apps where the applications consist of client-side(front-end) and the server-side(backend). The client-side is what the user see, whatever type of device it is, being a computer or mobile apps. The client-side responsibility is showing the data to the user in an easy to

read interface, along with taking their requests and passing it to the server. The backend consists of two primary components: application business logic and data processing/management. The data processing/management operates on various resources being users, persistent data, files, etc. The business logic manages triggering notifications based on changes in the data, prevent unauthorised access and figure 3.3 illustrates this.

3.4 API

For the applications to communicate with the back-end, there needs to be a standard-based protocol that defines how data flows. Thus combining the protocol with the data structure definitions creates an Application Programming Interface (API). The typical format used in most applications is called Representational State Transfer REST which is an architectural style and approach to communications used in web services development. REST provides a list of verbs such as GET, POST, DELETE in which request can be made using HTTP. The diagram 3.4 below illustrates this.

Figure 3.4: API [2]

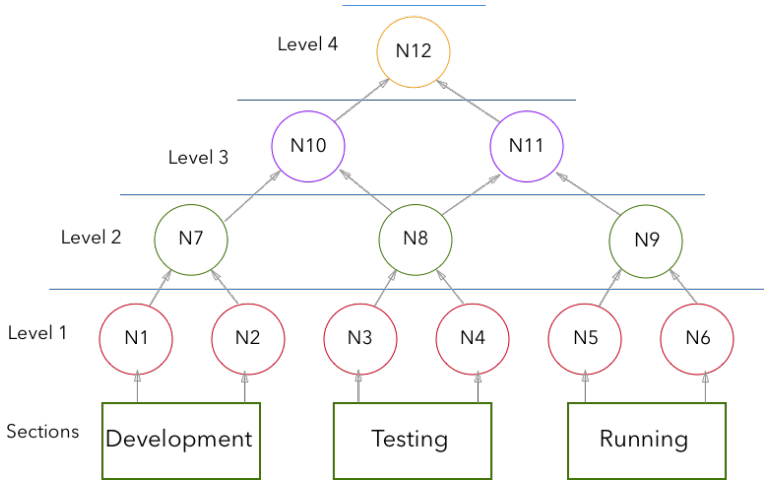


Chapter 4

Design

4.1 Methodology

Figure 4.1: Tree-Shape



After doing research, I decided to implement my methodology called Tree-Shaped. The idea behind this method is to en-corporate the functionality into different sections and prove each functionality at a lower, early stage. This benefits by setting out the goal into sections and filling in what functional requirements are needed to reach the target. The proving of functionality part is to able to fix any potential compatibilities at an earlier stage or remove if necessary.

The methodology has three parts

1. Sections

   The sections stage is the first step in using this tree-shaped. This part is where we set out what we are trying to achieve. Then incorporate each functionality into their respective sections.

2. Functionality

   After we define our sections including their respective functionalities, tests are to be written out for each functionality. The test shows that they still output the correct result, being what they are supposed to do.

3. Levels

   Levels stage is where we are proving/testing the functionalists. As we move up the tree into each node, the functionalists are combined. Thus solving compatibilities issues at an earlier stage of the project development.

4.1.1 Advantages

This methodology based on the test-driven development (TDD) process that relies on the repetition of very short development cycles. At each level, the nodes which hold the application are tested. These tests are from the functionality part when combining the previous nodes to make sure they still output the correct result.

4.1.2 Disadvantages

The methodology although sounds good from a theoretical point of view, but in a real world it has its drawbacks. Each node (circle in each level) requires making new test application, combine the previous two node applications and possible refactoring. This takes time what some projects do not have, but it reduces the number of bugs found by refactoring at each level. To overcome this, the methodology allows skipping one level to reduce testing times.

4.2 Functional Requirements

Functional requirement defines a function of a system or its component. After having discussions with outsourced developers and researching current systems, the following 4.1 illustrates the list of functional requirement. They are grouped into sections and what the aim of the project will deliver.

4.2.1 Development

Cloud Storage

Cloud storage is a service model in which data is managed remotely and made available to users over the Internet. This service allows developers to keep the application data in one or more locations, for the end users

Table 4.1: Functional Requirements

ID	Section	Name	Description	Priority
1	Development	Cloud Storage	Create, Read, Update, Delete objects	High
2	Development	Push notifications	Send push notifications to devices	Medium
3	Production	Analytics	Measure users in app activities	High
4	Production	Backup	Backup database to remote site	Low
5	Development	Self hosted	Host the MBaaS on developers server	High
6	Production	Remote Configuration	In app live updates	High
7	Production	A/B Testing	Testing different variations	High
8	Development	Live Database	Update objects without user refreshing	Low
9	Development	Dashboard	Interface for developers manage apps	High
10	Testing	Exceptions	Tools to collect and view exceptions	High
11	Testing	Test environment	Testing bugs/issues in a test enviornment	Medium

to access. Mobile app users want the ability to be connected to others, to share information but without filling up local storage. This cloud service solves this problem by storing the data remotely, and the application makes requests through some protocol to access the user's data. The service will provide tools for the developer to create, read, update and delete the data.

Sprint board

Challenges faced when developing an application, is trying to keep a list of features implemented in each version. There are already different sprint board applications out there, but wanted a way to en-corporate it all in one single dashboard application. This feature will be incorporated with exceptions discussed in section 4.2.2, where an exception can be attached to a ticket. This will help with tracking issues and signed off once completed.

Self hosted

This will allow the developer to host their system, to have full control of when running, and not to have to worry if the provider is going to shut down the service. By giving the developer a way to host their back-end then this will keep the cost down of not having to pay for third party services.

4.2.2 Testing

Exceptions

When an application crash, the reason for this is called an uncaught exception. An exception is an event, which occurs when the application is running, that disrupts the flow of a set of actions. An exception can occur when trying to read a value from a variable that does not exist. When developing applications, a good design is to handle all potential exceptions. This is known as a caught exception, and the application does not crash.

These exceptions can only be seen by the developer when testing the application, but can not always find these potential issues. This project will design a way, which the both uncaught and caught exceptions can send to the developer.

Test environment

In the exceptions section 4.2.2,we discussed that bugs/issues could occur will application, but testing the application on live data is the wrong approach. This can be done when the application is being tested; a test database will be used without the hassle of creating one. The cloud services currently do not provide a testing environment, where the integrity of the data is a priority. The project will be designed so that when the application is in debug mode, it will use the testing environment.

4.2.3 Production

Backup

During the research when asking developers what services are missing from current mobile cloud services, one was a backup feature. They wanted a way to completely backup their data being files and database contents. This would enable them to both have the piece of mind that their user's data is safe, and to have the option to change cloud service providers. It seems that providers create their systems in such a way that once the developer starts using it, they would not be able to migrate. This is down to not providing a service to transfer their user's data. This system will provide two services: one to backup their data to a remote or local location, and secondly to import data to aid in migrating to using this system.

A/B Testing

A/B testing also known as split testing is comparing two variations of a page to see which performs better. Currently this popular with web pages but my plan is to bring this to mobile applications. All mobile applications no matter what services they provide all have one goal: a reason to exist. A/B testing allows you to make more out of your existing traffic. This is achieved by sending our two variants (A and B) to similar visitors at the same time and use analytics to provide display what variation wins. Included in the research, a survey was conducted in section 2.2.3 to see what users felt about mobile apps. The results were that end-users do choose whether or not they will continue to use an app based on the content. So by using A/B testing service, we can quickly find out what they do and do not like. So how can this be implemented in mobile apps? This leads on to the next service.

Remote Configuration

Remote configuration is a service that lets you change the behaviour and appearance of the app without requiring a new build to be published. When using this service, the default properties will be what the developer put in the build. Then when the developer wants to make an update, this feature can be used to publish a new version. The mobile app can then perform the update in the background. Another feature is giving the user an option of choosing a theme for the app. This theme can change the look of the app from dark to a light mode for example. So what use remote configuration?. It can provide the following:

- Quickly roll out changes to your app
- Customise your app depending on the version they are running
- Use this along with A/B testing to find improvements

Notifications

Apple push notifications (APNs) provides the developer with a way to reach their users and perform tasks in the background. A powerful tool to keep the app in real time and users connected to the app. These notifications can be used along with the remote configuration service, to notify the users of a new theme out.

Analytics

Analytics to give the developer real-time information on the activity of their app, how users are interacting with the app. It can help understand the mobile app and users to evaluate the performance of the content. Analytics will also be used along with A/B testing already discussed in section 4.2.3.

4.2.4 Non Functional Requirements

4.2.5 Security

Security is a big part of any cloud-based applications. The user's personal information being sent up to the cloud, where potential hacks could expose these. The system will need to ensure its security and the integrity of its data.

Keys

The systems support two keys authentication; there is one key that allows requests to access the web server services. The next key allows data back and forth to the database. The development chapter will illustrate how this is being achieved.

Database

The mobile applications accessing the backend, will each have their database. This will provide data security to ensure users data do not get accessed accidentally by another app. These are only accessed once the app key is authenticated.

4.3 Deliverables

4.3.1 SDK

The software development kit (SDK) provides the developer with the necessary tools in the application to communicate with the web server (backend), through the API. The number of services discussed next will be included in the SDK.

Storage

Figure 4.2 illustrates the design of the storage library. The objects that the developers require for their app will all need the same functionality, being the way that the objects exchange information with the server. This then states that they all need to conform to the same way, and this is done using a protocol. The storage library will contain an object-relational mapping (ORM) tool, that will speed up parsing the objects to and from JSON format. This way the developer does not need to understand JSON format, but that the functionality is there to handle it. The last part of the design is that the properties being sent to the server will always be of type *String*. The reasoning for this, is that if the developer decides to change their mind regarding the client app object property, it will not ruin the already stored data. The *String* type is the only one that can handle all over types such as *Int* and *Boolean*. The development chapter will go into detail what protocol in software terms mean and how it is implemented within the storage library.

Figure 4.2: Storage SDK Design

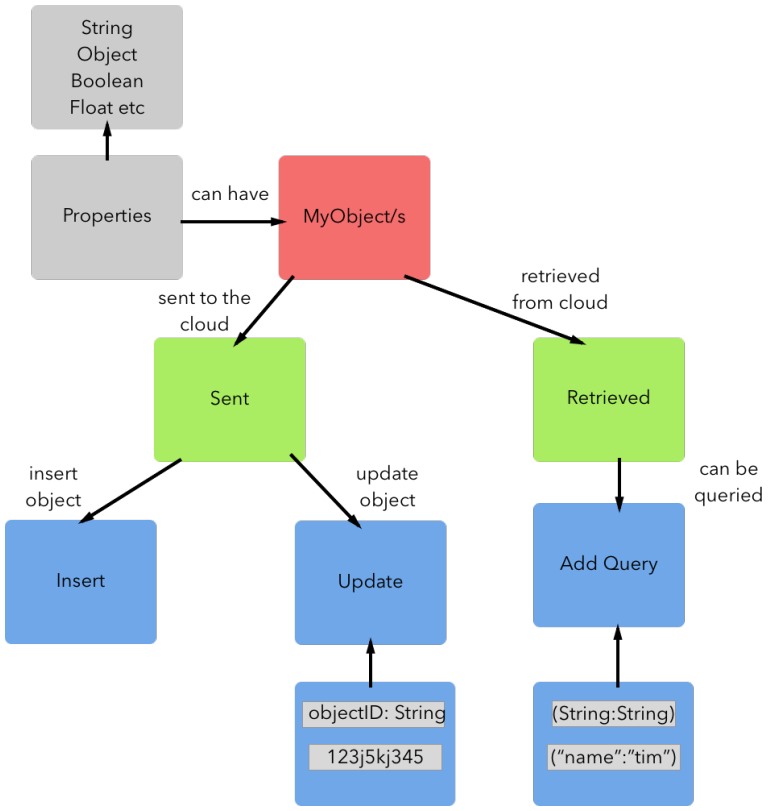
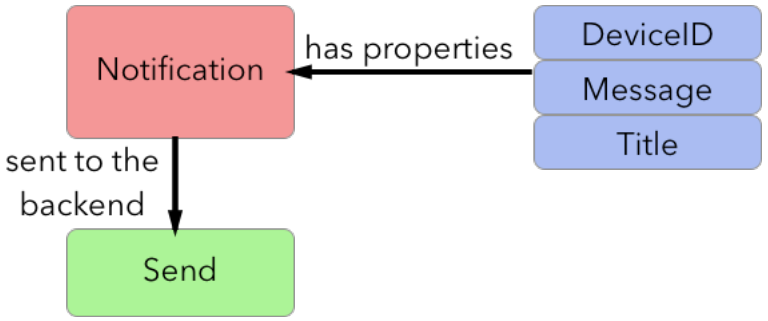


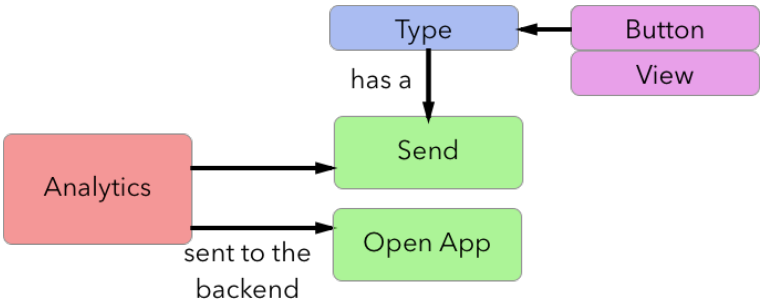
Figure 4.3: Notification SDK Design



APNs

Apple push notifications as stated above provides a way for apps to alert the user that an action has occurred. This action can be a text message or a friend that has been added. The notifications have to be activated by a sender, so when the user sends a message, a notification object will be sent to the server and in turn to the receiver. The library will contain a notification class, that will contain the required properties and the functionality to send the notification as illustrated in figure 4.3.

Figure 4.4: Notification SDK Design

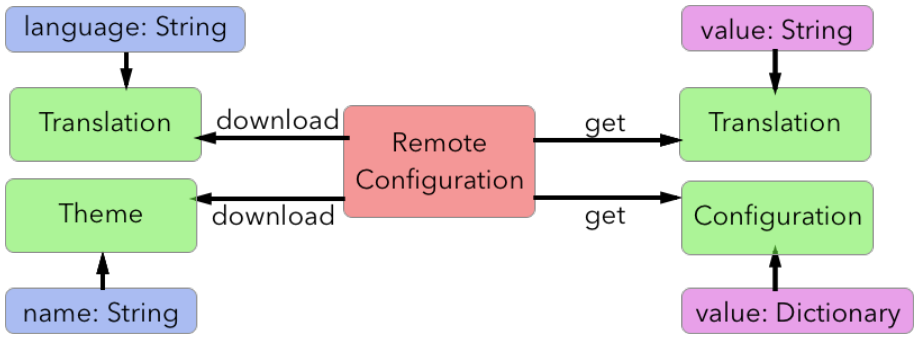


Analytics

The analytics library as illustrated in figure 4.4, has the capabilities of sending categorised types of analytics to the server. The type can be when a user clicks a button, or when a view is opened. The class will contain many different functionalities to make it easier for the developer to use. An example can be seen in 4.4 where the open app function is straight forward to send.

Remote Configuration/Language

Figure 4.5: RC/Language SDK Design



This section contains two parts: remote configuration and language. The are designed together as they both use the same system to update the application remotely. Figure 4.5 shows two components of the library, where one part is for downloading the correct language and configuration file, the other to retrieve from. The downloading section for both takes a key parameter which distinguishes it from the other such as language name, or theme name, i.e., Dark, Light.

The remote configurations and translations will be stored in JSON files, that can be both easily retrieved and stored on the device. Using the JSON files will speed up reading for each object properties. The structure of the configuration file as illustrated in 4.6 contains an object inside a controller class, and then the properties of the object.

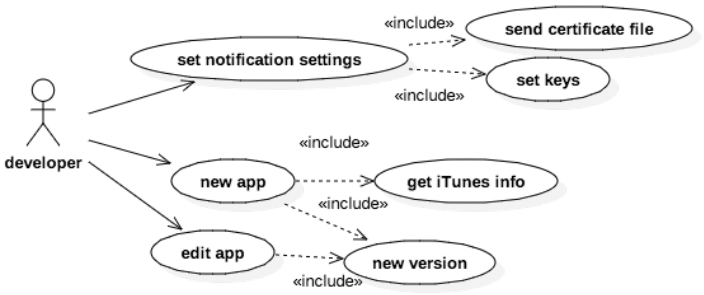
Figure 4.6: RC File Design

```
"controllers": [
  {
    "name": "ViewController",
    "parent": "0",
    "classProperties": {
      "backgroundColor": "blackColor"
    },
    "objectsList": [
      {
        "objectName": "tfName",
        "objectDesc": "",
        "objectProperties": {
          "type": "UITextField",
          "backgroundColor": "white",
          "fontSize": "",
          "isEnabled": "",
          "text": "",
          "textAlignment": "",
          "isUserInteractionEnabled": "",
          "isHidden": "",
          "placeholder": "",
          "textColor": "blackColor",
          "font": ""
        }
      },
```

4.3.2 Dashboard

Settings

Figure 4.7: Settings Use Case Diagram



The use case diagram for the settings views shown in 4.7. The developer in this view can set up the notifications requirements such as sending the certificate file. This certificate is crucial for sending notifications; it authenticates the developer's id when about to send the notification object.

The next part of the settings view, the developer can create and edit applications. These apps are the mobile apps that will be used in conjunction with the web-server and the SDK. These apps and versions will be used throughout the rest of the dashboard when setting remote configuration for example. When an application is being updated, some key values can also be retrieved from the iTunes API.

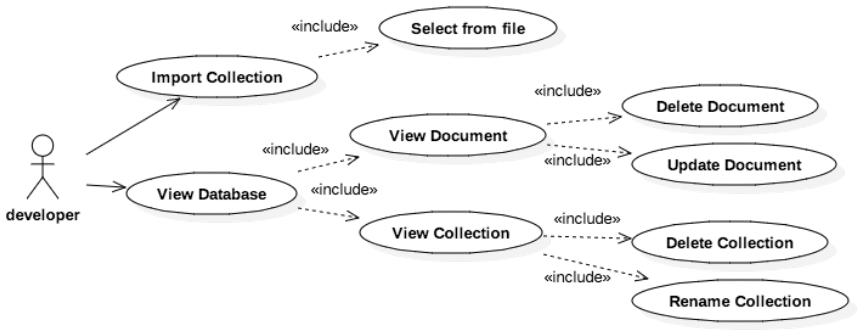
Storage

Figure 4.8 illustrates the storage/database use case. In the storage view in the dashboard, the developer will be able to choose a database which is specific to each application. After which be able to view and select the collections within and see the content records. Another feature is the ability to import collection from a JSON or CSV file into the database.

This use case is limited by design, the typical create, read, update and delete (CRUD) operations known with database development will not be available. When designing systems that include both backend and frontend, the most important one of two is the frontend. This is what the end users will see, so this system will focus on designing the structure of each collection in the application, in the development stage.

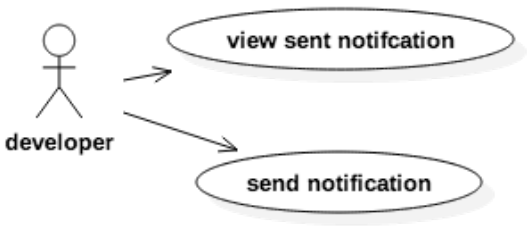
Typically the dashboard will allow the developer to perform CRUD operations, and then they will have to mimic that structure for the frontend app, thereby creating a potential bug. A bug can occur if someone can change the collection name, or collection property name, thus creating an inconsistency between frontend and backend. Removing the capabilities from one side being backend will hopefully eliminate this potential issue.

Figure 4.8: Storage View Use Case Diagram



Notifications

Figure 4.9: APNs View Use Case Diagram



Apple push notifications (APNs) that have sent can be viewed in the notifications view as illustrated in 4.9. The developer can also send notifications from the dashboard to the mobile apps, for example telling users about an

update, or new theme, etc.

Remote Configuration

Figure 4.10: RC View Use Case Diagram

The developer in the remote configuration view once an application and version have been chosen can create versions. These versions define the user interface of the mobile app. Each version will be dependent on an app version or a particular theme. In figure 4.10 illustrates what the capabilities of this view. After a version is created or being updated, the properties for the UI objects such as a label can be set.

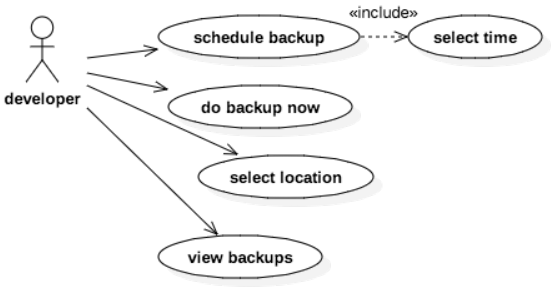
Languages

Figure 4.11: Language View Use Case Diagram

The language view use case in figure 4.11 is similar to the remote configuration use case above. This is because both are designed in the same way, that each version can be downloaded and the data is retrieved in the app.

Backup

Figure 4.12: Backup View Use Case Diagram

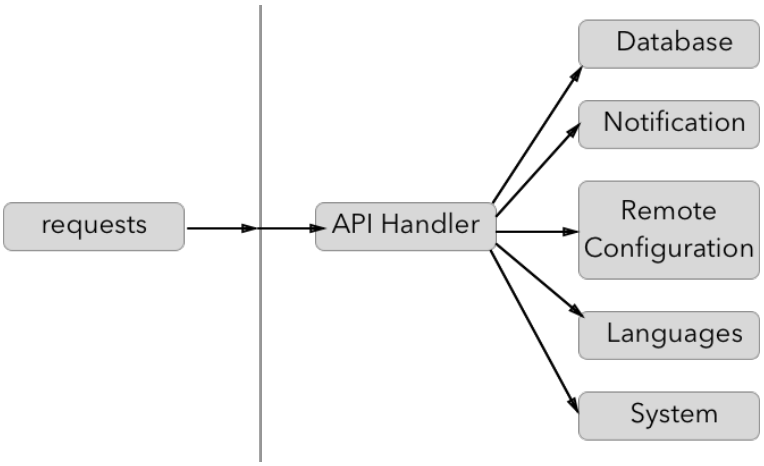


The backup view is where the developer can either set up a scheduled backup or do a backup now. In the use case figure 4.12, all the previous backups can be seen in this view. The developer will also be able to select a

location for these backups placed, being either remotely or locally. The backups will have the time-stamp and zipped to help save space.

4.3.3 Web Server

Figure 4.13: Web Server Design



The web server brief design is illustrated in 4.13. The web server uses an application programming interface (API) which defines a set of methods for communication. In this system, the database and notifications are an example of defined tools that will be used. The API Handler takes the request in and forwards it on to the correct web app to handle the request, and relay a response back. The development chapter under web server section will discuss in detail the list of tools.

4.4 Design Principles

Human-Computer Interaction (HCI) principles play a major role when designing an interface. These principles help keep applications of the same nature alike. An example is a mail app, the icon for a mailbox or sending messages can be used to convey without having to read a manual of what a button does. As this system will be using a Mac application, the macOS interface guidelines will be closely followed.

Apples Human Interface Guidelines [44] discussed the following design principles:

1. Mental Model

   - "..is the concept of an object or experience that people carry in their heads". It is the model of what users believes about the system, so the mental model of past experience on similar systems will be carried when looking to use this system. This will involve looking at current systems available and design the interface somewhat similar.

2. Direct Manipulation

   - ..is an example of an implied action that helps users feel that they are controlling the objects represented by the computer. When designing a view that the user can control, the objects such as deleting a record should only become invisible once the user has taken action.

3. User Control

   - ..principle of user control presumes that the user, not the computer, should initiate and control actions. The interface should give the user the control depending on the type of user. So a professional user will want more power compared to a novice user.

4. Consistency

   - "..allows the user to transfer their knowledge and skills from one app to another". So by designing the system will the same general layout of other apps will help the user not feel lost.

Chapter 5

Development

5.1 Introduction

As explained in the design chapter, this project includes three deliverables. So the development will include; a Mac app dashboard, Perfect web-server and CocoaPods framework. Using the tree-shaped methodology, the web-server was split up into their separate sections which include development, testing, and run. Then each service was design, developed and testing before moving up the tree. The development was broken up into phases, this way the project is kept on track on what has been completed and left to do.

1. Services Development
2. Integrate into Live App
3. Web server
4. Dashboard Development
5. CocoaPod Framework

As this project methodology is based on Test Driven Development (TDD) approach, the services were developed first and tested before adding to the project. Once these services have cleared testing individually, they were integrated into the DIT-Timetable app explained later, this was to ensure how the new services were developed would pass Apples pre-publishing tests. Once this was accomplished, the dashboard was designed next to view the services in an interface. Then lastly the services were developed into an SDK to be used in any mobile application.

5.2 Project Management

Good software project management is essential in developing and delivery of software projects. Software development is often difficult to estimate the time required to complete the project, especially when using new technologies. Project milestones can be used to monitor the development progress of the project at certain key points. Above includes the list of the key points within the project.

Each point had its time frame; this kind of project can lead to expanding to including more services and tools, so sticking to the list will help be on schedule.This project differs from a commercial project, in that the project manager, designer, developer and tester are all the same person. And with any project, testing plays a significant role in software development and can easily be left out.

5.3 Services Development

Each service had it's own or where possible shared Perfect server application along with Playground app. This made is easy to decide whether or not it was possible to implement each service into the project. For some of the services, the same Perfect web-server was used and able to be adapted to accommodate the requests needed.

Setup Before any development was started, some services were required to be setup. Perfect web-server developers provide an assistant app to help with the configuration and include any required packages needed to develop the API. The list of packages for the project includes the following:

1. https://github.com/PerfectlySoft/Perfect-Turnstile-MongoDB.git

   -Used to provide functionality to interact with MongoDB database, and provide authentication when requests come to the server.

2. https://github.com/PerfectlySoft/Perfect-RequestLogger.git

   -Provides the web-server with a logging system.

3. https://github.com/hkellaway/Gloss.git

4. https://github.com/PerfectlySoft/Perfect-Notifications.git

   -Aid with the push notifications.

5. https://github.com/PerfectlySoft/Perfect-SMTP.git

   -To add capability of sending e-mails

6. https://github.com/PerfectlySoft/Perfect-Zip.git

   -To zip backups folders, when sending to a remote location.

After the Perfect web-server was setup, the mongoDB database was required to installed locally. This was done by running the following commands in list 5.1

```
breew update
brew install mongodb
brew services start mongodb
brew services status mongodb
brew services stop mongodb
```

Listing 5.1: MongoDB setup

```
import XCPlayground
XCPSetExecutionShouldContinueIndefinitely(continueIndefinitely: true)
```

Listing 5.2: Playgrounds setup

The request over HTTP will be made using asynchronous; this does not block the interface while the request which can be time-consuming. The interface should not be blocked, making the user wait. To be able to run asynchronous requests in the playground apps, the following lines of code in Listing 5.2 is required to the top of the file.

Database storage

The database storage section was split up into two parts of the development.

1. Object-relational mapping (ORM)
2. Storage - how to send the objects to store persistently

ORM The database storage service required an object-relational mapping (ORM) which is a powerful method for designing and querying database models at the conceptual level, where the application is described in terms easily understood by non-technical database developers. It is a technique for converting data between incompatible type systems in object-oriented programming languages. The ORM was developed using Playground tool, where the creation of objects and parsing into JSON objects was done. The functionality of the ORM is to create a new object, parse it and send to the server, and be able to bring all objects back from the server.

This was developed using protocols and protocol extension. Protocols as Apple states "defines a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality." and extensions

are "new functionality to an existing class, .., protocol type.". [45] A protocol called *TBJSONSerializable* was developed as seen in Listing 5.3

```
public protocol TBJSONRepresentable {
    var TBJSONRepresentation: AnyObject { get }
}

public protocol TBJSONSerializable: TBJSONRepresentable {
    init( jsonObject : TBJSON)
    init()
}
```

Listing 5.3: Protocol

The first protocol *TBJSONRepresentable* just states that this variable *TBJSONRepresentation* has to be used in any class or protocol conforming to that protocol. Inside the *TBJSONSerializable* protocol, we have two methods that any class or structure used throughout any mobile application are required to used. The *TBJSON* to a type alias that is a dictionary, which holds the JSON objects which will be used when parsing. Next the protocol extension was developed, where the objects will be parsed into a dictionary type form that then can easily be parsed in JSON. In Listing 5.4 illustrates how the objects are parsed.

```
public extension TBJSONSerializable {
    var TBJSONRepresentation: AnyObject {
        var representation = [String: AnyObject]()
        for case let (label?, value) in Mirror(reflecting: self).children {
            switch value {

            case let value as Dictionary<String, AnyObject>:
                representation[label] = value as AnyObject?

            case let value as Array<CGFloat>:
                representation[label] = value as AnyObject?

            case let value as Array<String>:
                representation[label] = value as AnyObject?

            case let value as Array<AnyObject>:
                var anyObject = [AnyObject]()
                for ( _, objectVal ) in value.enumerated() {
                    var dict = [String:AnyObject]()
                    if let jsonVal = objectVal as? TBJSONRepresentable {
                        let jsonTest = jsonVal as! TBJSONSerializable
                        if let jsonData = jsonTest.toJSON() {
                            for (index, value) in convertStringToDictionary(text: jsonData) ??
    [String: AnyObject]() {
```

```
                                dict[index] = value
                            }
                            anyObject.append(dict as AnyObject)
                        }
                    }
                }
                representation[label] = anyObject as AnyObject?

            case let value as AnyObject:
                if let myVal = convertToStr(name: value) {
                    representation[label] = myVal
                } else {
                    if let jsonVal = value as? TBJSONRepresentable {
                        var dict = [String:AnyObject]()
                        let jsonTest = jsonVal as! TBJSONSerializable
                        if let jsonData = jsonTest.toJSON() {
                            for (index, value) in convertStringToDictionary(text: jsonData) ??
    [String: AnyObject]() {
                                dict[index] = value
                            }
                        }
                        representation[label] = dict as AnyObject
                    }
                }
            default:
                break
            }
        }
        return representation as AnyObject
    }
}
```

Listing 5.4: TBJSONSerializable

In Listing 5.4, the *TBJSONRepresentation* variable contains a switch case to loop through the mirrored object properties; which is a representation of the sub-structure. Then each property type such as String, Int or Dictionary, and depending on the type can be parsed into an AnyObject type and assigned to the dictionary with the key being the name of the variable.

The design also stated that in retrieving objects, functionality is to be in placed to parse back to objects. A design feature that was discussed in the design chapter, is that the values going up to the server are in *String* format. The functions to parse the object next will try parse the values from both their type or type *String*. A list of functions was implemented to parse each value back to the desired type using protocol extension to type

dictionary and method chaining. The method chaining speeds the development time for the developer, without needing to find the correct type first. An example of this is in Listing 5.5 which takes the parameter of the key and tries to parse the value in two ways. First, if the value if of type integer, it will return that value, but if it is of type String, it will convert a string to integer and return the new value.

```
func tryConvert(forKey key:Key, _ defaultVal :Int = 0 ) -> Int {
    var value = defaultVal
    if let int = self[key] as? Int {
        value =  int
    } else {
        guard let test = self[key] as? String else {
            return defaultVal
        }
        guard let integerVal =  Int(test) else {
            return defaultVal
        }
        value = integerVal
    }
    return value
}
```

Listing 5.5: Dictionary

Storage As part of the project architecture with regarding storage, any structure should have the functionality to send the object to the server, and retrieve without the need to create another function to setup and communicate over HTTP. As described in the design chapter, all structures conforming to the protocol will be able to send and retrieve the object/s between the server and the application. JSON is used in the transfer of data, a format in which both server and client can understand. JSON objects are simply just dictionaries where each value is an object of some type. The following table 5.1 illustrates the library commands to send and retrieve objects between the cloud storage. The *T* in the return column means that this function return type is of a particular type. In the following examples will be returning type *TBJSONSerializable* which is our protocol.

Table 5.1: My caption

Library Method	Description	Parameters	Result
getFilteredInBackground	Retrieves the filtered object	query: [String:AnyObject], type:T.Type, appKey: String = ""	T Object
getInBackground	Retrieves the object	objectID: String, type:T.Type, appKey: String = ""	T Object
removeInBackground	Removes the object	objectID: String, appKey: String = ""	Successful/ Error
sendInBackground	Update or send the object	objectID: String, appKey: String = ""	Successful/ Error

The next table 5.2 illustrates returning back array of objects.

Table 5.2: My caption

Library Method	Description	Parameters	Result
getFilteredInBackground	Retrieves the filtered object	query: [String:AnyObject], type:T.Type, appKey: String = ""	T Objects
getAllInBackground	Retrieves the object	objectID: String, type:T.Type, appKey: String = ""	T Objects

Figure 5.1: Storage Sequence Standard

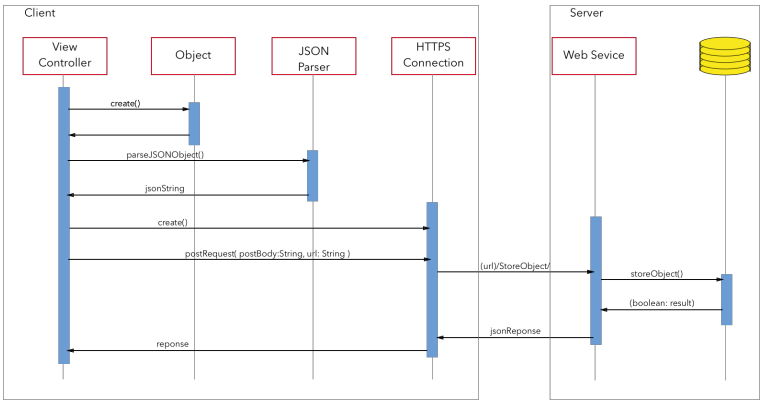
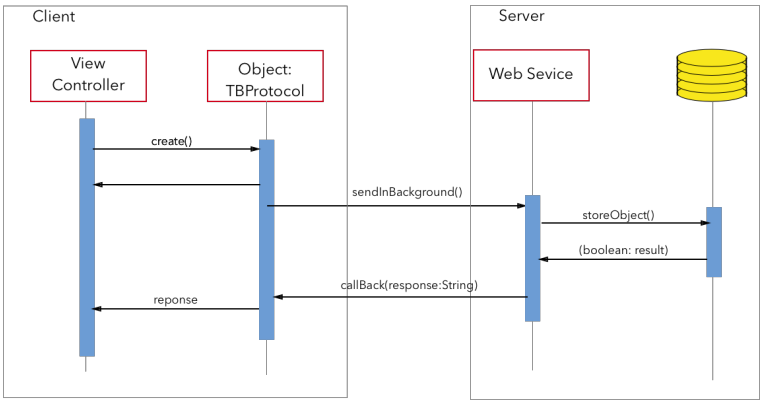


Figure 5.2: Storage Sequence New



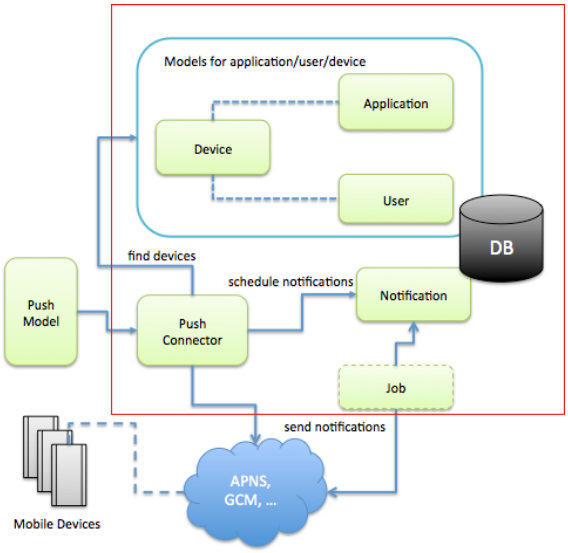
The two Figures 5.1 and 5.2 show the difference between developing the standard way, with the developing having to create a JSON parser and then set-up an HTTPS connection to send it to the server in Figure 5.1 . The new way in Figure 5.2 using the projects storage protocol involves the developer only having to create the object, then using the functionality of the object to send it to the server.

Apple Push Notifications (APNs)

Server The APNs uses JSON Web Tokens (JWT) to authenticate the credentials for the connection. This uses two keys, public and private where Apple where keeps the public and the private which is a .p8 file is used from the sender. In this system, the private key needs to be upload to the directory in which the web server

reads from. Several steps are required to accomplish this. The developers must have a developer account with Apple, then in the developer console, the .p8 key can be created and downloaded. After the web server test project required accessing that file to send notifications. The DIT-Timetable app was used to verify receiving the push notifications.

Figure 5.3: APNs [3]



In Figure 5.3 represents the stages in which to send push notifications. Inside the red box is the server layer, where the .p8 file is located, the push model is where the messages originate. The notifications go through Apple's sandbox APNs, then on to the mobile devices.

Client The APNs tool was implemented in the library, for which the developer can use the notification object to send a request to the server, which in turn sends the notifications. The notification's object requires a list of values to be set for the notifications to work.

- Universally Unique Identifiers - the target's device unique id to which apple will the notification.
- Message - message which the source wants to send to the target.
- Badge Number - each notification can be assigned a number, which displays on the app icon.
- Title - the title of the notification, usually the app name.

The following table 5.3 demonstrates notification library to send notifications.

Figure 5.4 shows how to use the API requests outside of the library.

Table 5.3: APNS Library

Library Method	Description	Parameters	Result
TBNotification.sendNotification();	Sends notifications object to server	None	Successful/ Error

Table 5.4: Analytics API Requests

API Call	HTTP Method	Description	Parameters
/{appKey}/notification	POST	send notification object	JSON Object
/{appKey}/storage/TBNotification	GET	Retrieves all notification objects	JSON Objects

Analytics

The first part for the analytic developed was the creation of the Perfect web-server which excepted POST requests. A playground application was developed with the lines of code in Listing 5.2 to aid with HTTP requests. This class gathers some information before sending the request including, time-stamp, build version, OS version, device make and model. For the purposes of testing this service, the data was hard-coded in separate function, that in turn would be reading from plist file. Some of methods and the parameters are included in the following Table 5.5 which shows how to use the library to send analytics. The server stores the analytic objects in the database corresponding to the application, this is done using the appKey which is sent up in the API request.

Table 5.5: Analytics Library

Library Method	Description	Parameters	Result
TBAnalytics.sendOpenApp();	Sends up object with open app type	view: UIView , method: String? = #function , file: String? = #file	Successful/ Error
TBAnayltics.send();	Send up object with type as option	app: UIResponder, type: SendType , method: String? = #function , file: String? = #file	Successful/ Error
TBAnayltics .getAllInBackground();	Retrieves all TBAnayltic objects	NONE	Array of TBAnayltics

Figure 5.6 shows how to use the API requests outside of the library.

Table 5.6: Analytics API Requests

API Call	HTTP Method	Description	Parameters
/{appKey}/storage/TBAnalytics	POST	Uploads analytics object	JSON Object
/{appKey}/storage/TBAnalytics	GET	Retrieves all analytic objects	JSON Objects

Remote Configuration

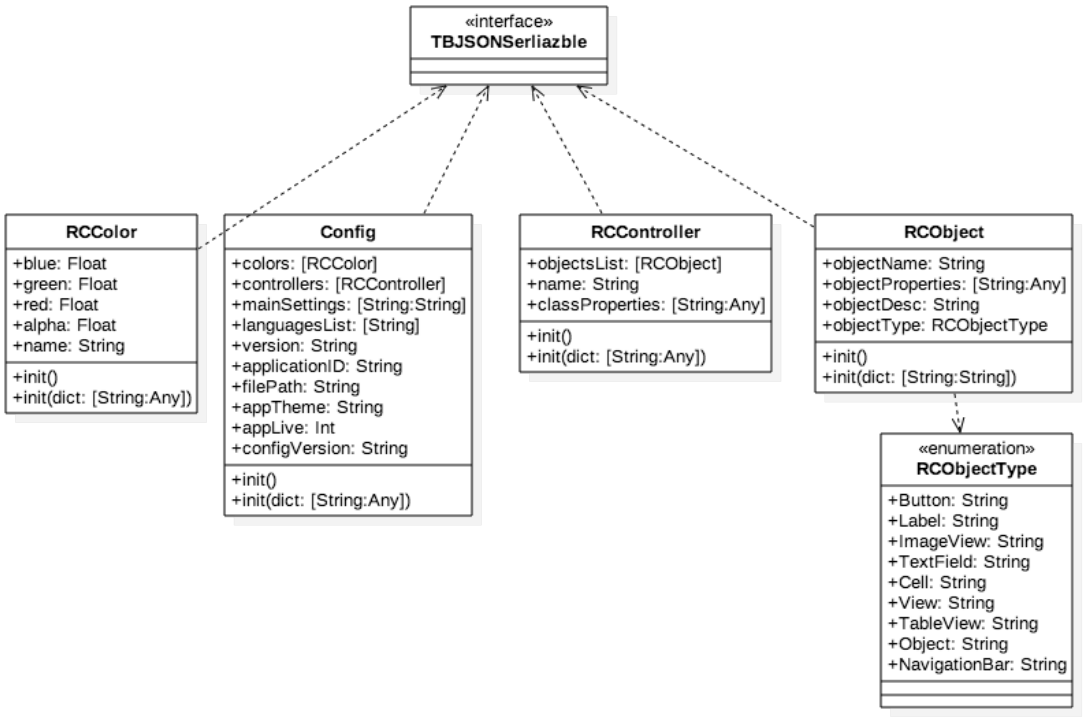
The remote configuration service development is broken up into four sections.

- JSON files - where the configuration objects will reside on the phone
- JSON file manager - how the retrieving of the values
- Objects configuration - how each interface object can be configured
- Remote updating - updating the configuration file

This section will discuss the first three, the remote updating will be explained in dashboard development section.

JSON files The first part was the development the JSON file layout. The design chapter already discussed the design of the remote configuration, where each class object will contain the objects relating to that class, and subsequently, the objects properties will be on the object. To help with this, the storage protocol *TBJSONSerializable* was used to be able to parse the objects into JSON string. In Listing 5.4 illustrates the class diagram for the complete remote configuration structure.

Figure 5.4: Remote Config Class Diagram



In Figure 5.4, the remote configuration structure consists of 5 classes. The very first class of the structure is the Config class and is where all the other class objects will be located. The *Config* class holds many properties, the first being the colours value, where all colours used in the application will be stored. The *RCColour* object contains the red, green and blue (RGB) values along with alpha and the name of each colour. This name will be used in the properties value of each object, for example, the background colour. Next is controllers of type *RCController*, these are the main classes or commonly known as *ViewControllers* in the mobile app.

Moving on to the *RCController* class, the properties include the name of that class which is the same variable name in app code The class properties, for example, if that class is of type *UIViewController*, that class will contain *backgroundColor* property, etc. The last major property is *objectList* of type *RCObject*. This is where all objects such as labels, text fields, tables, etc. will be stored. The *RCObject* class contains the name of the object, along with its properties, these are what are provided by the framework when developing an app. The object type property is a type of enumeration which will contain what base class name is, for example, *UILabel*. The reason for this will be explained later in the dashboard development section.

Back to the *Config* class, the main setting variable is key value pairs that hold the primary data such as URLs values; these are stored at this level to speed up retrieval. The language list is as the name states contain the list of translation languages that the end user can change. This will be explained later in the language service section.

JSON file manager Table 5.7 illustrates some of the main functions available in the library. The function performs a read on the JSON files based on parameters being passed in and returns a value to be used. This functions can be utilised by the developer throughout the mobile app and also used in next section for the objects configurations.

Table 5.7: JSON file manager

Library Method	Description	Parameters	Result
RCConfigManager.getColor();	retrieval of colour	name: String, defaultColor: UIColor	UIColor
RCConfigManager .getTranslation();	retrieval of translation value	name: String, defaultName: String	String
RCConfigManager .getMainSetting();	retrieves main setting value	name: String, defaultName: String	String
RCConfigManager .getObjectProperties();	retrieves object properties	className: String, objectName: String	[String:AnyObject]
RCConfigManager .getConfigVersion();	gets latest version of config file	None	call back method
RCConfigManager .getConfigThemeVersion();	gets latest version of config theme file	None	call back method

Objects configuration Object configurations involve how to get the object properties and update the user interface (UI) object. In the design chapter, it was discussed that a protocol along with protocol extension would be used for each UI object. A separate protocol for each UI object will be developed. A protocol is first defined, then an extension to that protocol to add the implementation. A snippet example of *UILabel* object is in the following Listing 5.6

The extension *LabelLoad* is restricted for classes with type *UILabel*, and the developer has two methods it can use to implement. Inside the setup function, the object properties are retrieved from the JSON files in line 12 and then set to the corresponding property value in lines 15 and 18. The implementation chapter demonstrates how this labelling protocol can be used.

```
1  public protocol LabelLoad {}
2  public extension LabelLoad where Self: UILabel {
3
4      public func setupLabelView( className: UIViewController, name: String = "") {
5          self.setup(className: String(describing: type(of: className)), objectName: name)
6      }
7      public func setupLabelView( className: UIView, name: String = "") {
8          self.setup(className: String(describing: type(of: className)), objectName: name)
9      }
10     private func setup( className: String, objectName : String ) {
11
12         let dict = RCConfigManager.getObjectProperties(className: className, objectName:
       objectName)
13         for (key, value) in dict {
14             switch key {
15             case "text" where dict.tryConvert(forKey: key) != "":
16                 self.text = RCConfigManager.getTranslation(name: viewName)
17                 break
18             case "backgroundColor" where dict.tryConvert(forKey: key) != "":
19                 self.backgroundColor = RCConfigManager.getColor(name: (value as! String),
       defaultColor: .white)
20                 break
```

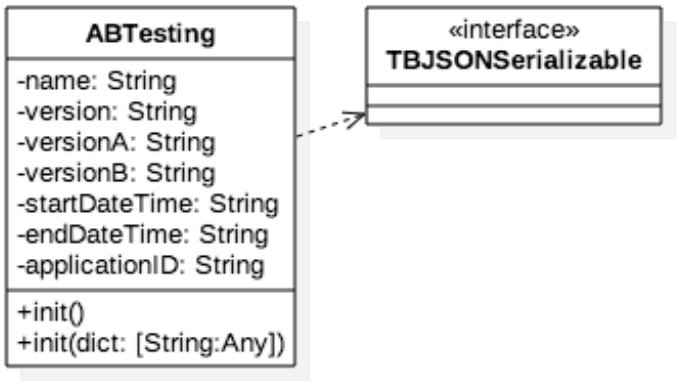
Listing 5.6: UILabel Protocol

A/B Testing

A/B Testing utilises two other services, remote configuration and Analytics. The JSON files can tell what version of configuration is being used, so when an analytic object is sent up, the version is included. The server, however, needs some development to handle A/B Testing. The dashboard explained later is used to set what mobile applications and versions will be involved in an A/B Testing. When a request comes into the web server for the configuration file, a check on the A/B Testing list is done to check whether that app version exists as seen in Figure 5.5.

The class diagram comprises of the *ABTesting* table along with the protocol TBJSONSerializable mentioned earlier. The table properties include the name of the testing, the application id and app version for that testing.

Figure 5.5: A/B Testing Class Diagram



The *versionA* and *versionB* are the two different configuration files and the dates for which the testing is in place.

To develop this service on the server, a singleton class (which is a class that has only one instance in the life of an app) *RemoteConfig* is used. When the server starts up, the singleton class is initialized with a current request number of zero. With each request, the count increases, and using this value can depend on what version of the configuration file that users get. The web servers section illustrates the singleton using a sequence diagram later.

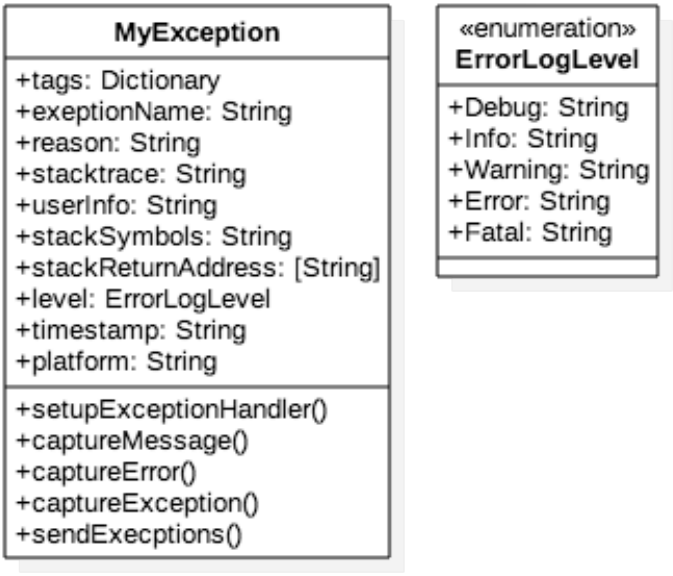
Exception catching

The design chapter explained there are two types of exception catching, uncaught and caught exceptions. Both types need to develop in different ways, as one would potentially crash the app, so would not able to send a POST request to the server. Not only are there two types of exceptions, but each exception has a different level, so when the developer views the dashboard, they can set a priority. The various levels are Fatal, Error, Warning, Info and Debug.

The exception was developed using a singleton class to that all exceptions can be sent through. When the app opens, the exception objects gets initialized which includes setting the *NSSetUncaughtExceptionHandler()* which is Apples uncaught exception handler, where the parameter is an internal function name which looks after catching the exception. The following Figure 5.6 illustrates the exception class. The tags properties hold values relating to the device type, the OS running, etc., the rest of the values are about the exception values.

Uncaught exceptions Uncaught exceptions make the app crashes, so between the time crash happens and the app closes which is a small window, the exception has to be dealt with. Using the *NSSetUncaughtExceptionHandler()* with the parameter of the function, that function is not allowed to make outside calls, which

Figure 5.6: Exception Class Diagram



include HTTP request and function calls. So the exception is stored in *UserDefaults* which is a built-in the data dictionary that stores a few user settings for as long as the app is installed. When the user opens application again, the exception singleton class gets instantiated, and it does a check if any exceptions exist in *UserDefaults*, and then an HTTP post sends the exception to the server.

The Listing 6.12 shows the code required to catch uncaught exceptions. On line 21, Apples function takes a parameter which has to be an internal function name. The internal function called *exceptionHandler* takes a parameter of type *NSExeception*, which contains values such as the reason and the call stack. As this function can not make outside requests, its stores the values in UserDefaults storage. When the application opens again, the function on line 2 gets called which reads the stored values and sends them to the server.

```
1  public func setupExceptionHandler() {
2      checkAndSendErrors()
3
4      func exceptionHandler(exception : NSException) {
5
6          #if DEBUG
7              NSLog("Name:" + exception.name.rawValue)
8              if exception.reason == nil {
9                  NSLog("Reason: nil")
10             } else {
11                 NSLog("Reason:" + exception.reason!)
12             }
13         #endif
14
```

```
15          UserDefaults.standard.set(exception.name.rawValue, forKey: "name")  //Integer
16          UserDefaults.standard.set(exception.reason ?? "Nil", forKey: "reason") //setObject
17          UserDefaults.standard.set(exception.userInfo ?? "Nil", forKey: "userInfo" )
18          UserDefaults.standard.set(exception.callStackReturnAddresses , forKey: "
        stackReturnAddress")
19          UserDefaults.standard.set(exception.callStackSymbols , forKey: "stackSymbols")
20      }
21      NSSetUncaughtExceptionHandler(exceptionHandler)
22  }
```

Listing 5.7: Exception handling

Caught exceptions Extra functionality was added to using the same class as above with uncaught exceptions to enable developers to send these exceptions to the server. As the class is a singleton and gets initialized when the application opens, the *sharedClient* function can be used to return the instance of that object, then use the functions available to send the caught exceptions. Some of these are shown in Table 5.8

Table 5.8: Caught Exceptions

Library Method	Description	Parameters	Result
MyException. sharedClient. captureMessage();	Captures the info message and sends to the server	message : String, method: String? = #function, file: String? = #file, line: Int = #line	Successful/ Error
MyException. sharedClient. captureMessage();	Captures the message along with error level to the server	message: String, level: ErrorLogLevel, method: String? = #function , file: String? = #file, line: Int = #line	Successful/ Error
MyException. sharedClient. captureError();	Captures the error and sends to the server	error : NSError, method: String? = #function, file: String? = #file, line: Int = #line	Successful/ Error

5.4 Integrate into Live App

For some parts of the project, an already developed and published app called DIT-Timetable was used to add in the services, to test if Apple would allow it through. The services include remote configuration and language choice. Due to Apple's strict guidelines, the remote configuration was developed into the DIT-Timetable app into different phases, then each stage had a build and published.

Phase 1

This phase included just the basic remote configuration, with the capability of updating text such as page title, and label values. Apple did approve this phase meaning that the app using the iPad prototyping app can be updated.

Phase 2

Phase 2 gave the ability to adjust user interface values such as text colour, text size and user interaction enabling/disabling. This also has been approved by Apple giving it a go ahead to be completely integrated into the project.

5.5 Web-server

The web server developed was split up into two sections; the development of the web-server using the framework Perfect with setting up the server and creating the installation file which will install all the required dependency packages.

Development

To start developing the server, Perfect provides a basic template with just the structure to start off with, found at https://github.com/PerfectlySoft/PerfectTemplate. A new directory was created, then running the command: *git clone* `https://github.com/PerfectlySoft/PerfectTemplate.git`. One the template was download, running the following command would create the Xcode project: *swift package generate-xcodeproj*.

A package manager tool is for managing the distribution of Swift code. It automates the process of downloading, compiling and linking dependencies. The package consists of the source files and a manifest file called *package.swift*. It defines the packages names and contents using the *PackageDescription*. [46] The package.swift file required updating with the required packages such as MongoDB and Notifications etc, then running the command *sudo swift build* inside the project directory retrieved all the packages that were included in the package.swift file. The project structure contains sources and packages directories, the packages are which was already downloaded and the sources where the web-server files are placed.

The web-server starts off with the *main.swift* which includes the creation the HTTP server, and adding routes and setting the port number, then starting the server. The routes are where each REST request goes, so for example if request /user then the route will go to the user class. An example of the routes is shown in listing 5.8. This example is the routes for the database, retrieving and sending object. The function is called from the *main.swift* file, which returns all the routes for the database handler class. The handler parameter is the method name in the same class for which implementation is done depending on the route.

```
1 public func makeDatabaseRoutes() -> Routes {
2     var routes = Routes()
3     routes.add(method: .get, uri: "/api/{appkey}/storage/createIndex/{collection}/{index}/",
      handler: mongoCreateIndex)
```

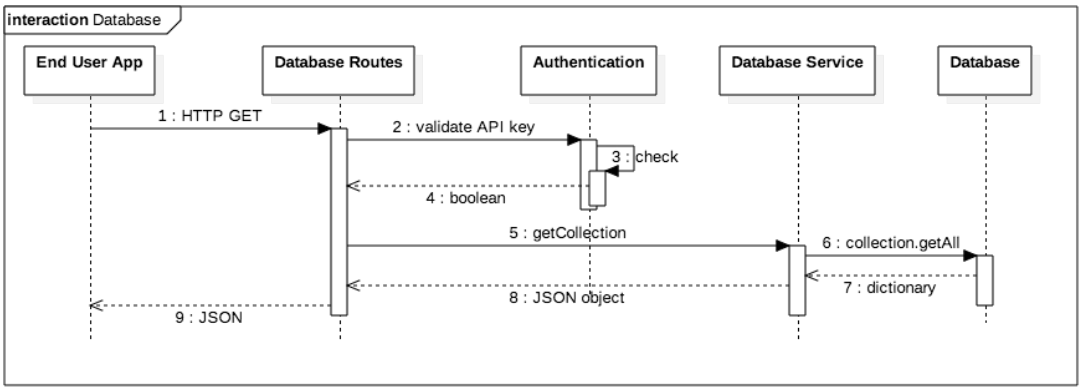
```
4     routes.add(method: .get, uri: "/api/{appkey}/storage/dropCollection/{collection}/",
      handler: mongoDropCollection)
5     routes.add(method: .get, uri: "/api/{appkey}/storage/dropIndex/{collection}/{index}/",
      handler: mongoDropIndex)
6     routes.add(method: .get, uri: "/api/{appkey}/storage/rename/{oldcollection}/{newcollection
      }", handler: mongoRenameCollection)
7     routes.add(method: .get, uri: "/api/{appkey}/storage/{collection}", handler: mongoHandler)
8     routes.add(method: .get, uri: "/api/{appkey}/storage/{collection}", handler: mongoHandler)
9     routes.add(method: .get, uri: "/api/{appkey}/storage/{collection}/{skip}/{limit}", handler
      : mongoQueryLimit)
10    routes.add(method: .get, uri: "/api/{appkey}/storage/{collection}/{objectid}", handler:
      mongoFilterHandler)
11    routes.add(method: .post, uri: "/api/{appkey}/storage", handler: databasePost)
12    routes.add(method: .post, uri: "/api/{appkey}/storage/{collection}", handler:
      databaseCollectionPost)
13    routes.add(method: .post, uri: "/api/{appkey}/storage/all/{collection}", handler:
      databaseCollectionsPost)
14    routes.add(method: .delete, uri: "/api/{appkey}/storage/{collection}", handler:
      removeCollection)
15    routes.add(method: .delete, uri: "/api/{appkey}/storage/{collection}/{objectid}", handler:
       removeCollectionDoc)
16    routes.add(method: .post, uri: "/api/{appkey}/storage/remove/{collection}/{objectid}",
      handler: safeRemoveCollectionDoc)
17    routes.add(method: .post, uri: "/api/{appkey}/storage/query/{collection}/{skip}/{limit}",
      handler: databaseGetQuery)
18    routes.add(method: .post, uri: "/api/{appkey}/storage/query/{collection}/", handler:
      databaseGetQuery)
19    routes.add(method: .post, uri: "/storage/{collection}", handler: databaseCollectionPost)
20    return routes
21 }
```

Listing 5.8: Routes

The web-server deliverable consists of a collection of apps. These apps is a web application that does something e.g. database of records. The design chapter under web-server section 4.3.3 discussed the overview structure, and what web apps are to be developed. This section will discuss the development of each web app and what functionality is required.

Database The database app consists of four layers; the routes, authentication, service and the database as can be seen in figure 5.7 This allows the app to be loosely coupled meaning the database type can be changed without having to change the other layers. The objective of this app is to perform the four basic functions of persistent storage; create, read, update and delete (CRUD). So the first four routes were developed, and the

Figure 5.7: Database Sequence Diagram



collection name was required to be passed as a parameter as this is a generic app, the collection names would not be known until the developer creates them. This is show in Table 5.9, where the app key is also passed to be authenticated.

Table 5.9: Database routes

Route	HTTP Method
/api/{appkey}/storage/{collection}	GET
/api/{appkey}/storage/{collection}	POST
/api/{appkey}/storage/{collection}	DELETE

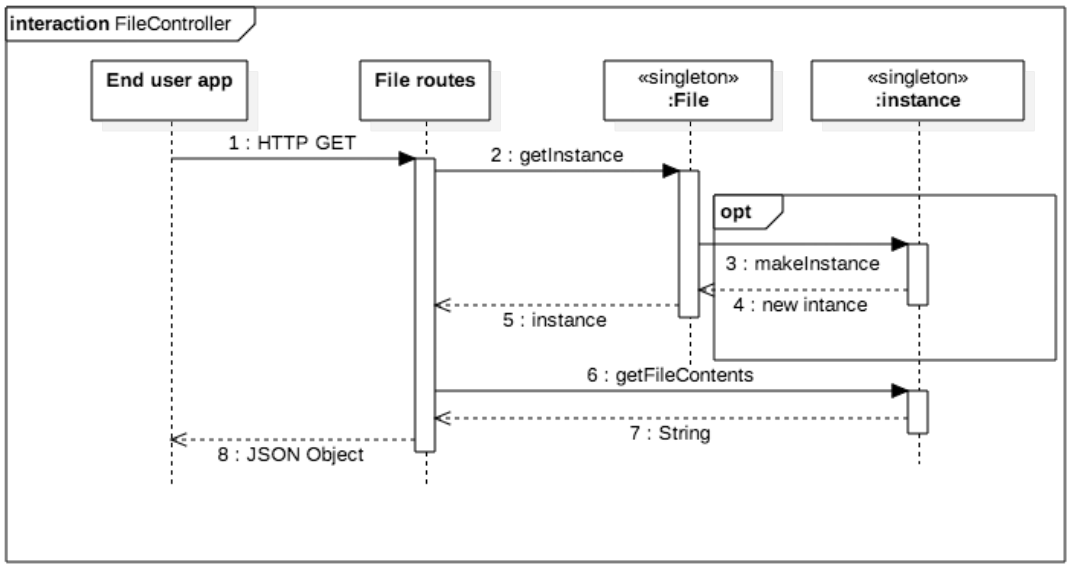
The database service layer function is called depending on the route; the service layer will then communicate with the database class to perform the task. The database class performs five steps for each of the CRUD operations;

- Open client connection - the connection being the Uniform Resource Identifier (URI) : "mongodb://localhost:27017"
- Database - connect to correct database depending on the application
- Collection - get the collection from the collection name which is passed in
- Find - run the query and pass in anywhere clauses if required
- Close - close the collection, then database and finally the client

Once these steps have finished, the function returns the search records, which in returns parses the records into JSON form to be sent back in the response object. After the four basic functions were completed, extra functionality was added to be able to pass query dictionary from the mobile application for example *"name":"tim"* which will return all records where name key contains values equal to *tim*. This was developed by creating another route along with function which accepts the dictionary query and then includes this in the find function. Next, the routes had to be added for the dashboard to make requests, for example, to retrieve all databases

and in turn all collections relating to each database. This is used in the database page called Storage already discussed above.

Figure 5.8: File Sequence Diagram



File This web app provides two purposes, one for uploading of files of any type and storing from the mobile app, as well as other web apps such as remote config discussed next to retrieved a stored file. The Figure 5.8 illustrates the steps required to manage files. This web app class using a singleton pattern. The difference between singleton class and regular class is that only one instance is made upon run time, this helps controller file writing so that two instances are not trying to write to the same file at the same time.

Table 5.10: File Handler Routes

Route	HTTP Method
/api/{appkey}/upload/{directory}/	POST
/api/{appkey}/upload/{filepath}/	GET

The routes to handler send and retrieving files as shown in Table 5.10. The post route sends up the files to a particular directory path which is set from the client application side, as well as the file names. After the file handler has successfully saved the files in the specified directory, the file controller class calls the database service class and inserts the file's meta-data that has been upload as history records.

This structure for the file web app allows the access from other apps such as the remote configuration and language to access the singleton class. Inside the class, there are a collection of functionalities required such as setting the working directory. This is another reason for using the singleton pattern, as every time the working directory is called, it creates a new directory which would cause files to be lost in a multitude of sub-folders. The instantiation of the file class happens when the server starts up in the *main.swift* file, thus given all over

apps access to one working directory.

Extra functionality was required if a file path does not exist, then the directories need to be created. An example of this would be if file path were *'/files/cars/image1.PNG'*, and the cars directory currently did not exists then steps is required to ensure the file location was correct.

1. check if the path exists - if true then overwrite
2. if path does not exit - get directory path from file path string = *'/files/cars/'*
3. create file path
4. save file to location

Figure 5.9: Remote Config Sequence Diagram

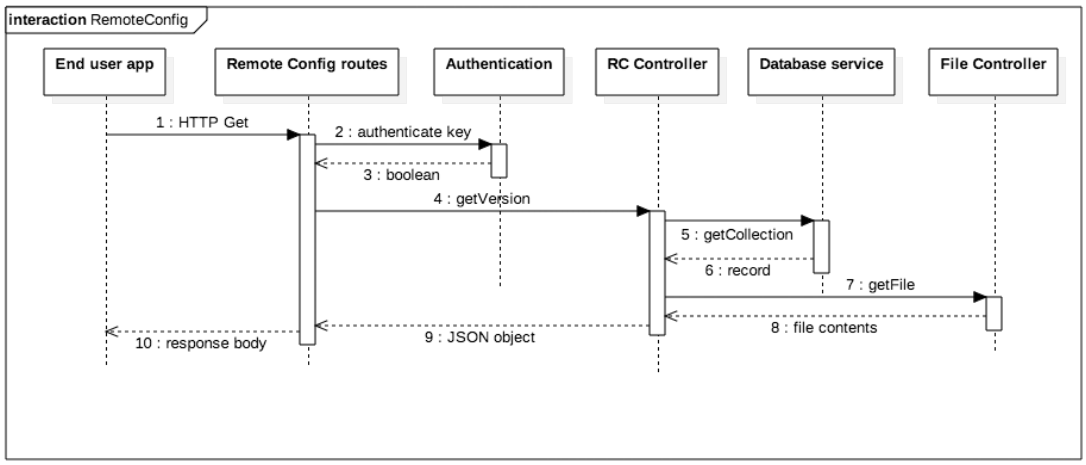


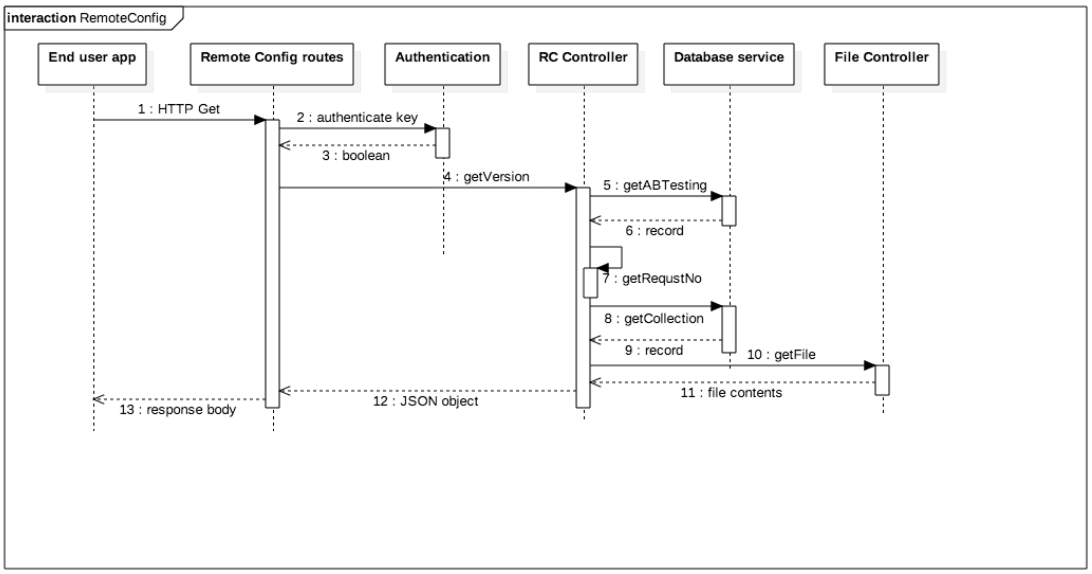
Table 5.11: Remote Config Routes

Route	HTTP Method
/api/{appkey}/remote	POST
/api/{appkey}/remote/{version}	GET
/api/{appkey}/remote/{version}/{theme}	GET

Remote Configuration As illustrated in Figure 5.9 operations are required to retrieve the configuration file contents. The HTTP GET request with the right route seen in table 5.11 is made, then the app key is again authenticated if passed the *RCController* class is called to get a version file. The *RCController* makes a call to the database service class for the collection of remote configuration along with the query of the app version. The record is returned to a file path; then file controller class is called to return the contents of the file.

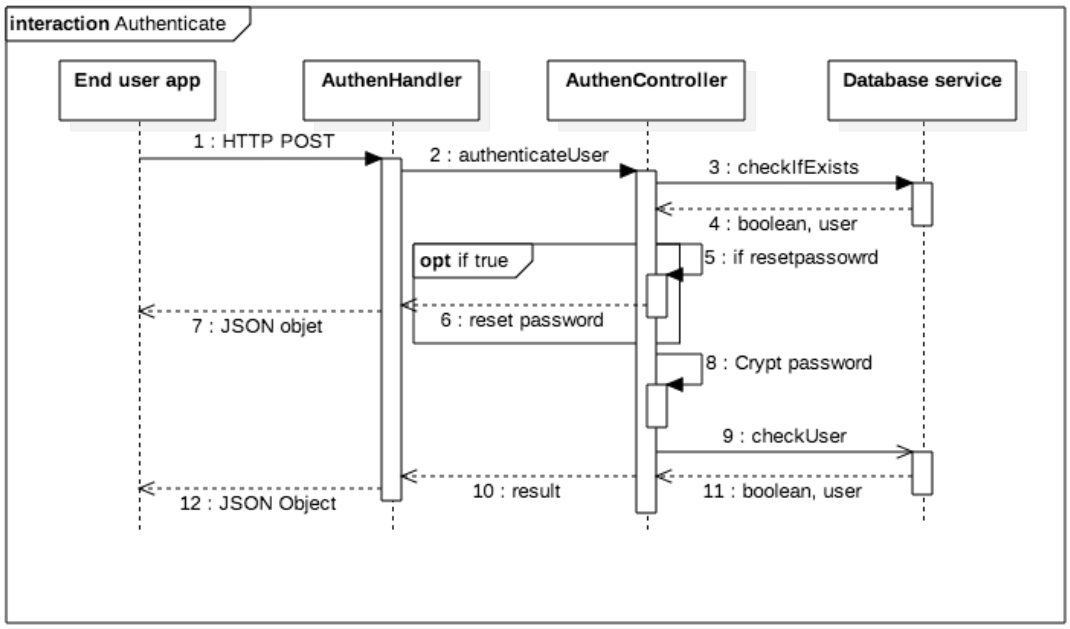
A/B Testing The A/B Testing service discussed in the services section above requires extra functionality, this is illustrated in Figure 5.10. An additional query to the database service was added to *RCController* class

Figure 5.10: A/B Testing Sequence Diagram



to get the records if any of A/B Testing. The A/B Testing records contain the application version which is queried from the version passed in, if the return contains a record, then a request to itself is made to get the request number. This request number is the total number of requests made modular 2, so depending on the number depends on what version A or B is returned to the response body. The *RCController* class is a singleton, meaning it only has one instance, and this was used to store the request no, instead of writing to a text file which would cause performance issues.

Figure 5.11: Authentication Sequence Diagram



Authentication Figure 5.11 illustrates the required steps to authenticate a user when signing in. The HTTP request method is POST to keep both username and password out of URL parameters for security purposes. The authentication handler class which handles the routes passes the values to the authentication controller class which performs four steps. The first step is to ensure the user exist using the username, next it does a check if the password reset field has been set to 1 meaning the administrator has authorised this account for the password to be changed. If the result is true, the message is sent back to the client app for the developer to allow the user to change the password. If the reset password flag has not been sent, then the password is first encrypted using the salt value used when registering.

The salt value is stored in a separate table for security reasons and is a one-way hashing function. The primary reason for using salt is to defend against dictionary attacks, historically a password would have been stored in plain-text, but over time additional steps were in placed to safeguard the password and one these ways is called Salting. After the password has been salted, then the query can be run by passing the username and hashed password, and if there returns one record, then the client app is sent an authentication message.

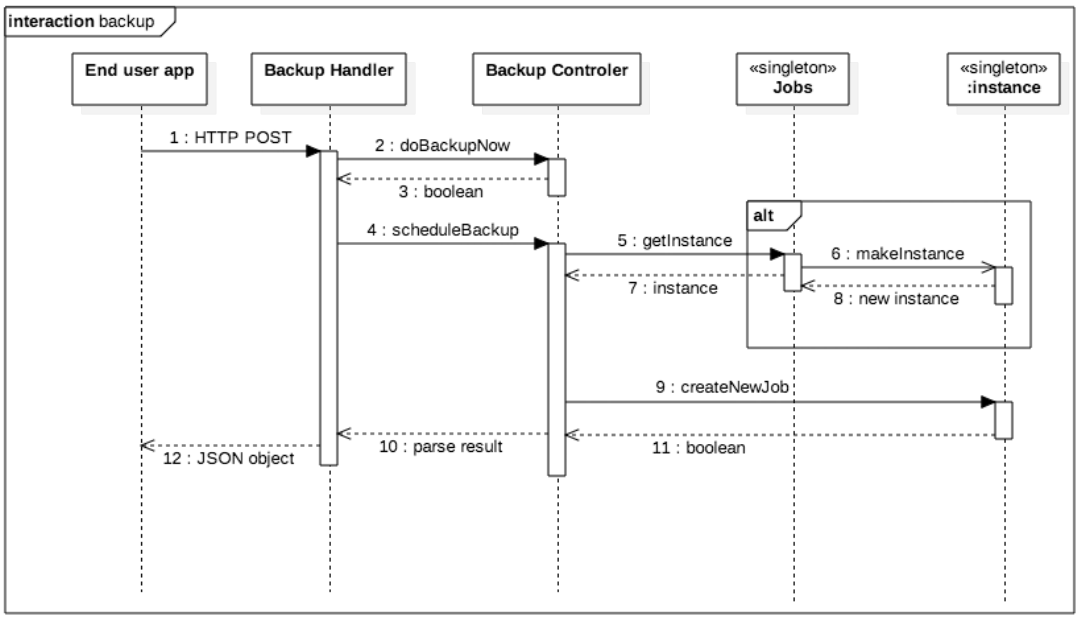
System The system handler and controller class are for the dashboard status view, the data for the four pie charts being CPU, memory and storage usage. This data is retrieved by running Linux commands from the server. The overview structure of this web app includes the system handler which only handles a get request for retrieving all the required data. The system controller class contains functions for each pie chart data, which is shown in Table 5.12 along with the command and output.

Table 5.12: My caption

Function name	Linux command	Values retrieved
getCPUUsuage()	top -bn1	user, system, idle
getMemoryUsuage()	free -m	total, used, available
getStorageUsuage()	df -h	size, used, available

The CPU usage is retrieved by running the *top* which displays all Linux tasks; the *-b* option puts the top in batch mode operation. The batch mode stops top from accepting any input until top has exited.The *n1* is the number of iterations or frames to be run before ending, in this case, one. The output is then parsed into a dictionary for user, system and idle. The *free* command displays the amount of free and used memory in the system. The *-m* option displays the output in MBs format. The result creates a list of columns, and the string is parsed and the total, used, and free values are appended to the dictionary. The *df* command gets a report of the disk space usage, and using the *-h* options returns the result in human readable format, e.g., 1K, 512M 2GB. After the three commands are run, the result is sent back to the response body parsed in JSON format to be read by the status view class.

Figure 5.12: Backup Sequence Diagram



Backup The backup feature discussed in the dashboard sections requires the web-server to do a complete backup of all database collections and files to a remote or local location. The sequence diagram in Figure 5.12 illustrates the steps required to accomplish this. The sequence diagrams display two types of requests if a backup is required to be done now or scheduled for a later time. The first is quite simple with the request coming in and passed to the backup controller class, and performs the required steps to accomplish the backup will be discussed after.

First, scheduling a backup requires a few more steps, the Jobs class is a singleton which keeps an instance of the *CronJob* class. *CronJobSwift* is a package provided by a developer called Ryan Collins with Github link `https://github.com/rymcol/SwiftCron-Example`. The *CronJob* is on the same principle of Linux software utility called *Cron* which is a time-based job scheduler. These jobs can repeatedly be run at certain times of the day, week, etc. to perform tasks in the background. The *CronJob* class still requires to be kept running, and outside of the main.swift file to be updated with new jobs or cancelling jobs. The singleton pattern is used to maintain an instance of *CronJob* running, and allow outside requests to update the jobs.

Back to the sequence diagram in Figure 5.12, the make instance method is only a fail-safe if the main file when the server starts up does not create the instance. When the instance is return back to the backup controller class, the job can be created and updated. All scheduled jobs are also stored in the database along with the job id, so that a request can be made to cancel a particular job.

Once the backup method has been called, a number of steps are required along with a package called Zip provided by Perfect. A Linux package is also required by running the following command *apt-get install libminizip-dev"*

The package for Swift zip can be found at the Github link `https://github.com/PerfectlySoft/Perfect-Zip`. The zip package will be used to put all the collections into one zipped folder so that it can store locally or remotely without potentially using up a lot of space.

1. get list of database
2. get list of collections from each database
3. stored the collections in a backup folder, sub-folded the collection name
4. zip the backup folder with the current date
5. move the zip folder to destination
6. put history of the backup in the database

Figure 5.13: Translation Sequence Diagram

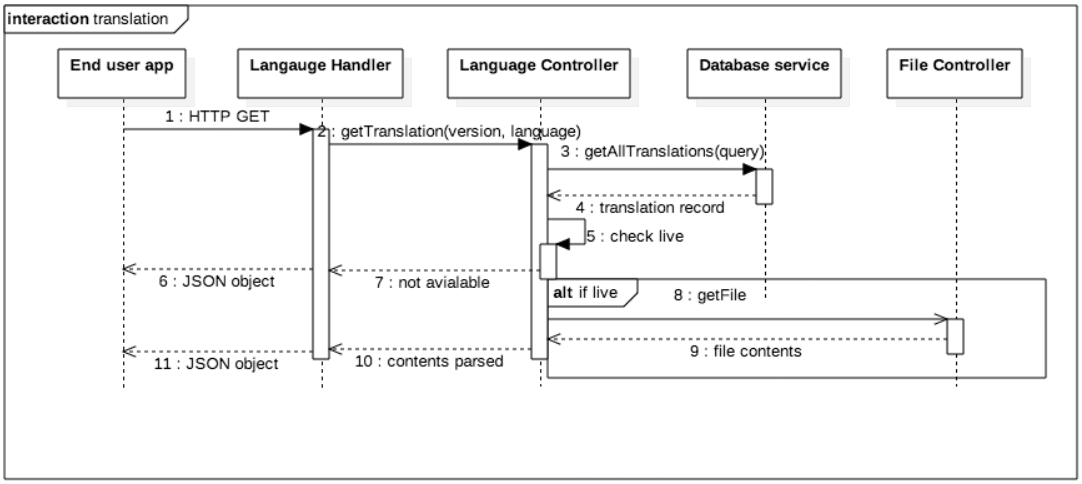


Table 5.13: Translation Routes

Route	HTTP Method
/api/{appkey}/translation	POST
/api/{appkey}/translation/{language}/{version}	GET
/api/{appkey}/translationVersion/{language}/{langversion}	GET

Translations The last web app to be developed is the translation, which handles posting and retrieving the correct translation version and language. The Table 5.13 illustrates the routes, where the first post request will send the translation object to both the database and file location. When a request is made for a translation file, an extra check is done to see if that version is available; this can be seen in the sequence diagram in Figure 5.13. This was discussed in the dashboard section, where the developer has the option of making a translation version live.

The translation and configuration files availability are put into the top object of any database GET request sent to the server. Listing 5.9 displays the header of an object when making a request, the current version and date of what is available on the server for both configuration and translation; then the application can make a request for the latest if required.

```
{
    "config": {
        "version": "1.0",
        "date": "12/01/2017"
    },
    "translation": {
        "version": "1.0",
        "date": "12/01/2017"
    }
}
```

Listing 5.9: Header JSON

Testing environment The last part of the web server work was to develop a way to enable a testing environment. This was done using the library with each request being sent up to the server, included in the header was a value that is to set if the developer is in testing/debug mode. Then on the server side, a new database would be created automatically to handle storing the test data. The storage view in the dashboard has a new functionality where a replication of a database can be done with a naming convention of <database name>-Test. This provides the developer with live data without having the worry of destroying valuable users data.

Deployment

DigitalOcean was used as the server of choice and to create a virtual private server or as DigitalOcean calls them Droplets. After creating an account and going to the page to create a droplet, the first choice was of distribution. The project required Ubuntu 16.04, after which the droplet size was asked, as for this project the basic 512MB ram, 20GB SSD Disk and 1000 GB transfer package was chosen. The next option was what region the droplet will be located and decided to go with London being the closest. Step 5 was additional options where IPv6 was chosen, for Apples requires the server to contain both IPv4 and IPv6. Step 6 was setting up SSH keys, which was done, and the last step was to give the droplet a name for the dashboard purposes.

After the droplet had been created and to log in from the computer, the terminal was used with the following command: *ssh root@<droplet IP address>*. Once the password had been entered, a prompt message asking to set up SSH keys and followed by entering YES. After logging in a list of steps to setup and install the required

packages, while running these commands and checking they installed correctly, they were added to a script file which would be used to create the installation script.

Step 1 The first step once logged in was to create a new user and password for security purposes. This was done by running the following commands in Listing 5.10.

```
useradd -d /home/USERNAME -m -s /bin/bash USERNAME
echo "USERNAME ALL=(ALL) NOPASSWD:ALL" >> /etc/sudoers

passwd USERNAME
```

Listing 5.10: Setting user-name

Step 2 Next, the server required to set the locale along with updating and upgrading.

```
export LC_ALL="en_US.UTF-8"
export LC_CTYPE="en_US.UTF-8"
sudo dpkg-reconfigure locales
sudo apt-get update
sudo apt-get upgrade -y
```

Listing 5.11: Updating Server

Step 3 To be able to run Perfect which is Swift based, Linux server requires the Swift package.

```
cd /usr/src
sudo wget https://swift.org/builds/swift-3.0.2-release/ubuntu1604/swift-3.0.2-RELEASE/swift
    -3.0.2-RELEASE-ubuntu16.04.tar.gz
sudo gunzip < swift-3.0.2-RELEASE-ubuntu16.04.tar.gz | sudo tar -C / -xv --strip-components 1
sudo rm -f swift-3.0.2-RELEASE-ubuntu16.04.tar.gz
```

Listing 5.12: Swift Installation

Step 4 The MongoDB database for persistent storage also required being installed.

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv EA312927
echo "deb http://repo.mongodb.org/apt/ubuntu xenial/mongodb-org/3.2 multiverse" |
sudo tee /etc/apt/sources.list.d/mongodb-org-3.2.list

sudo apt-get install -y mongodb-org
sudo systemctl start mongodb
sudo systemctl status mongodb
```

Listing 5.13: MongoDB Installation

Step 5 The supervisor is a service that can be installed to monitor and manage a program that is defined in the configuration file.The following Listing 5.14 will set up GIT which is where the web-server will be installed, and then the supervisor will be installed and set-up.

```
1  mkdir -p {running,app/.git} && cd app/.git
2  git init . --bare
3  cd hooks && rm -rf *.sample
4
5  #add the contents to file
6  nano post-receive
7
8  chmod +x post-receive
9
10 supervisord --version # check if installed
11 sudo apt-get install supervisor -y
12 service supervisor restart
13
14 cd /etc/supervisor/conf.d
15 sudo nano THE_APP_NAME.conf
16
17 sudo supervisorctl reread
18 sudo supervisorctl reload
```

Listing 5.14: Supervisor

At line 6 where the *nano* command is run, the *post-receive* file need to be updated to contain the project name.

Step 6 Next, the web-server itself needs to pull from Github and then built. Once the build has been done, ownership for the folders needs to change. Then the built packages need to be moved into the running folder, which supervisor will pick up the server and start it.

```
1  git clone https://github.com/collegboi/PerfectServer.git
2
3  cd PerfectServer
4  sudo swift build
5  sudo chown -R $USERNAME.$USERNAME .build/ Packages
6
7  sudo supervisorctl stop MyAwesomeProject
8  sudo cp -rf PerfectServer/.build/debug/* running/
9
10 sudo chown -R $USERNAME.$USERNAME running
11 sudo supervisorctl start MyAwesomeProject
```

Listing 5.15: Web-Server

Step 7 For the web-server to be access from the outside the server, a server needs to be installed to route the requests to the web-server. Nginx is installed, and then some configurations need to be done. When running line 3 in Listing 5.16, the port number needs to be updated to 8181 which is what the web-server runs on. Nginx will route traffic coming in at 80 and 443 which is HTTP and HTTPS to port 8181.

```
1 cd /etc/nginx
2 sudo mv nginx.conf nginx.conf.backup
3 sudo nano nginx.conf
4
5 sudo /etc/init.d/nginx restart
```

Listing 5.16: Nginx Server

5.6 Dashboard Development

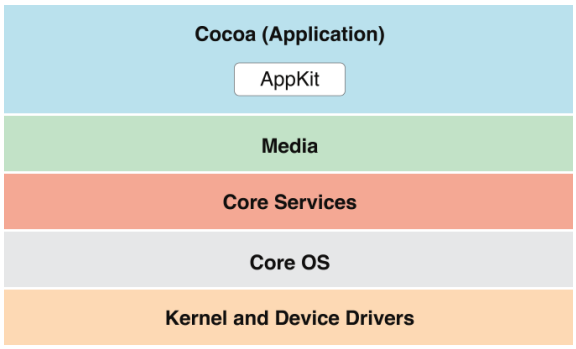
The dashboard was originally going to be created as an iPad app, but after some thought that not all mobile developers can be expected to own an iPad, the dashboard was developed as a Mac App. In the design chapter, the layout and design of the dashboard were discussed including what functionality will be provided to the developers.

5.6.1 Project structure

Xcode IDE was used to develop the dashboard interface with the help of libraries. These libraries included Cocoa(API) and Charts. Cocoa is Apples native object-oriented application programming interface (API) for their operating system macOS. The cocoa consists of the Foundation Kit, Application Kit and Core Data frameworks. It is responsible for the appearance of apps and their responsiveness to user actions. The Figure 5.14 illustrates where the Cocoa frameworks reside. Charts is a third party framework provided by a developer called Daniel Cohen Gindi that can be found on Github. [47]

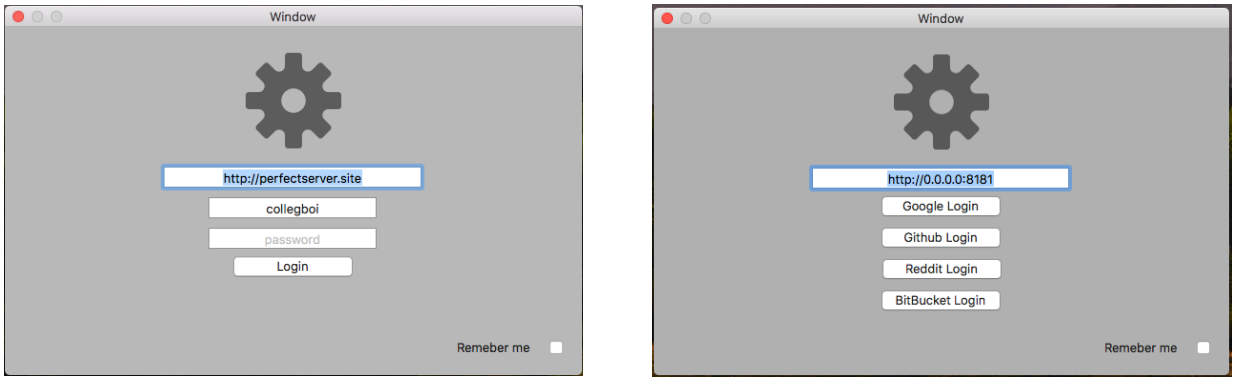
Already discussed in the design chapter, Apple gives an extensive section on Human Interface Guidelines on which to follow when developing Mac applications. So following these guidelines will help develop an application that can be submitted to the Mac App store. The IDE that will be used as already stated is Xcode, inside Xcode, several views can be utilised. In the following sections, the two main areas which will be used are the Interface Builder and Code Editor views. The Interface Builder is where the storyboards can be edited. They are a user interface way of designing and developing the UI of an application, and the code editor view is what connects the UI view to the class files.

Figure 5.14: Cocoa [4]



5.6.2 Dashboard Views

Login view



(a) Login View 1 (b) Login View 2

Figure 5.15: Configuring Apps

Figure 5.16: Log in class diagram

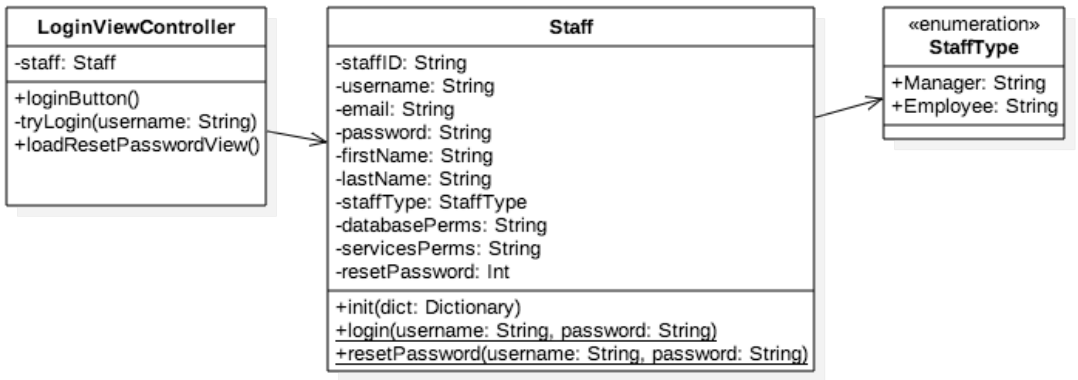


Figure 5.16 illustrates the class diagram of the main view controller for the login view, as well as the staff object that was used to authenticate the user. In Figure 5.15a illustrates the simple dashboard sign in view where the user gets verify their credentials. The view contains three input values which the user must put in, as the server can be deployed on any server, the user is asked to put in the IP or domain name where the server is

located. The other two values are user-name and password, and then there is an option for the user to tick the remind me box, which keeps the user logged in. The Log-in button sends POST request to the server along with both values. A register button was decided against, as this is a restrictive application. The administrator once logged in can create new users explained later on.

Initially, the plan was to use the login view in 5.15a and is still an option to revert to. The other way of authenticating a user, and in the industry is preferred to be used OAuth. The first step was to download the package from Github using *git clone –recursive https://github.com/p2/OAuth2App.git* inside the project directory. OAuth 2 provides a highly secure authentication service, and the Listing 5.17 shows the code necessary to perform this. The loader is a protocol which all services uses, which calls the *requestUserdata* function to do the login. The services class, for example, Google each contain their API endpoint for authentication; `https://accounts.google.com/o/oauth2/auth`. If the call back function is successful, then limited users information is stored locally to use within the application.

The view was updated as illustrated in Figure 5.15b, where several service options can be used.

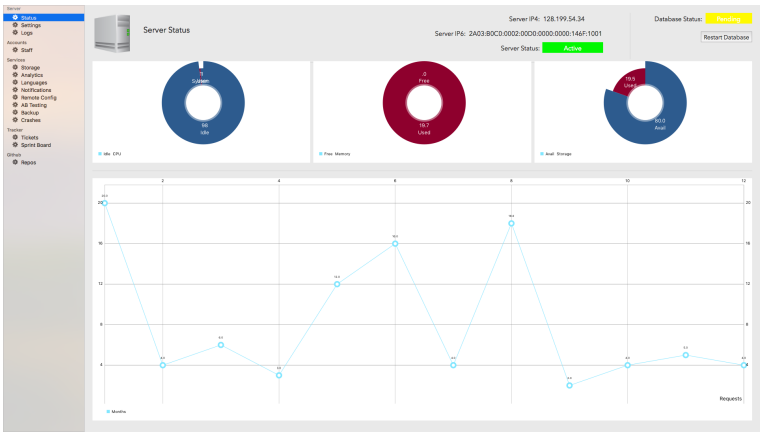
```
func openViewControllerWithLoader(_ loader: DataLoader, sender:  NSButton) {
    loader.oauth2.forgetTokens()
    loader.oauth2.authConfig.authorizeContext = view.window
    NotificationCenter.default.removeObserver(self, name: OAuth2CallbackNotification, object:
    nil)
    NotificationCenter.default.addObserver(self, selector: #selector(self.handleRedirect(_:)),
     name: OAuth2CallbackNotification, object: nil)
    loader.requestUserdata() { dict, error in
        if let error = error {
            //inform user of error
        }
        else {
            //stores data locally from dict
            self.loadMainView(sender: sender)
        }
    }
}
```

Listing 5.17: OAuth 2 Login

Status

Before discussing the status view, the menu on the left will exist on view. The menu bar routes the user across the whole application. The bar contains grouped submenus to help clarify to the user what the view provides. The accounts section, for example, provides staff, so before clicking on this option, it is already known that it

Figure 5.17: Status View



is staff accounts. This design is following Apple's guidelines on the mental model and consistency mentioned in the design chapter.

Figure 5.17 illustrates the server status view, it is the first page the user sees when entering the application. This view displays a number of graphs about the server. To help with displaying graphs, a framework was used called Charts, and listing 5.18 illustrates the code necessary to display a pie chart. In lines 1 to 4, the content of the chart is being populated. The values past are the sections within a pie chart, being the size of section and label name. The sections in line 9 are given their own color to help distinguish them.

The top section of this view, displays the current status of the server and Mongodb database. The server status and IP labels retrieves the data from digital ocean's API. The settings discussed next will show the configuration needed to make the requests to the API. The last part on the top right outputs the status of the database, giving the developer a button to restart the Mongodb instance if required. Starting from left to right with the pie charts, the first displays the current CPU usage, next storage usage and memory usage. The line chart displays the history and current CPU usage. This gives the developer an general overview on what the physical sever doing. This can help decide whether or not to upgrade the system.

```
1 for i in 0..<dataPoints.count {
2     let dataEntry = PieChartDataEntry(value: values[i], label: dataPoints[i])
3     dataEntries.append(dataEntry)
4 }
5 let pieChartDataSet = PieChartDataSet(values: dataEntries, label: label)
6 let pieChartData = PieChartData(dataSets: [pieChartDataSet])
7 pieChartView?.data = pieChartData
8 //assign colours for parts of the pie
9 pieChartDataSet.colors = colors
```

Listing 5.18: Pie Chart

The bottom line chart in figure 5.17, displays the number of requests made over a period of time.

Settings

Figure 5.18: Settings View

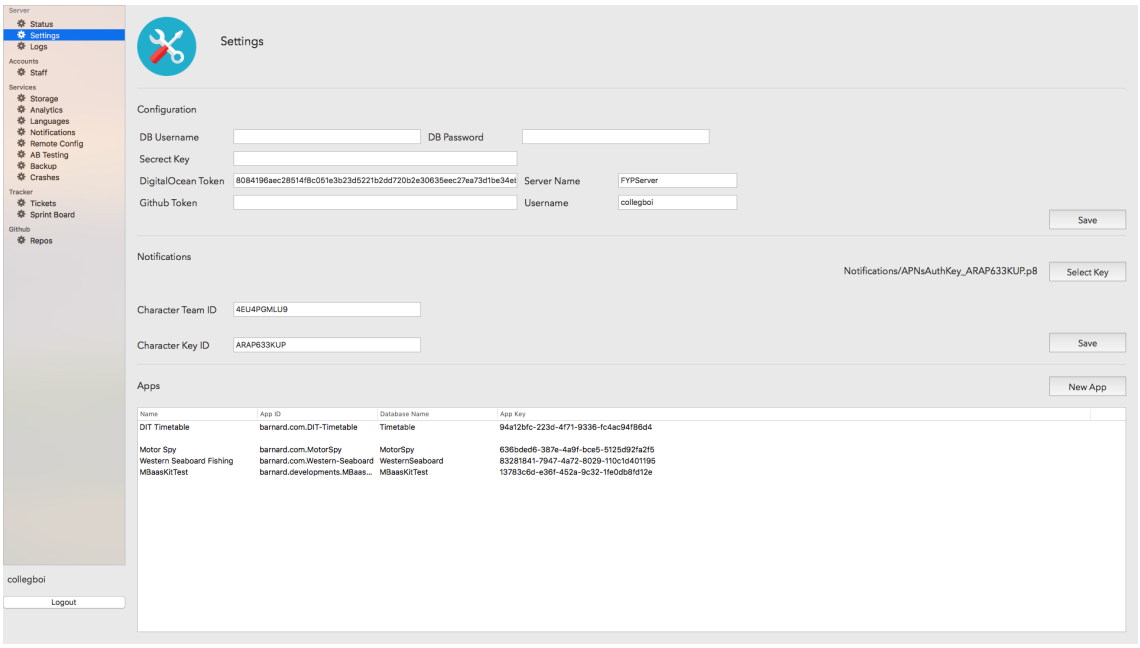


Figure 5.19: Apps Class Diagram

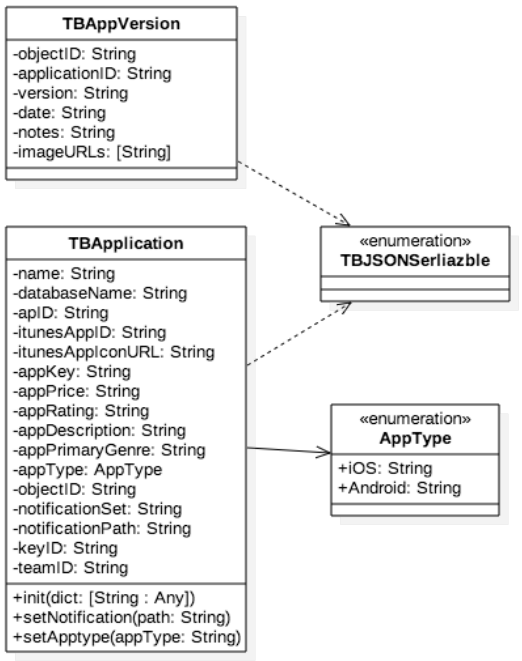
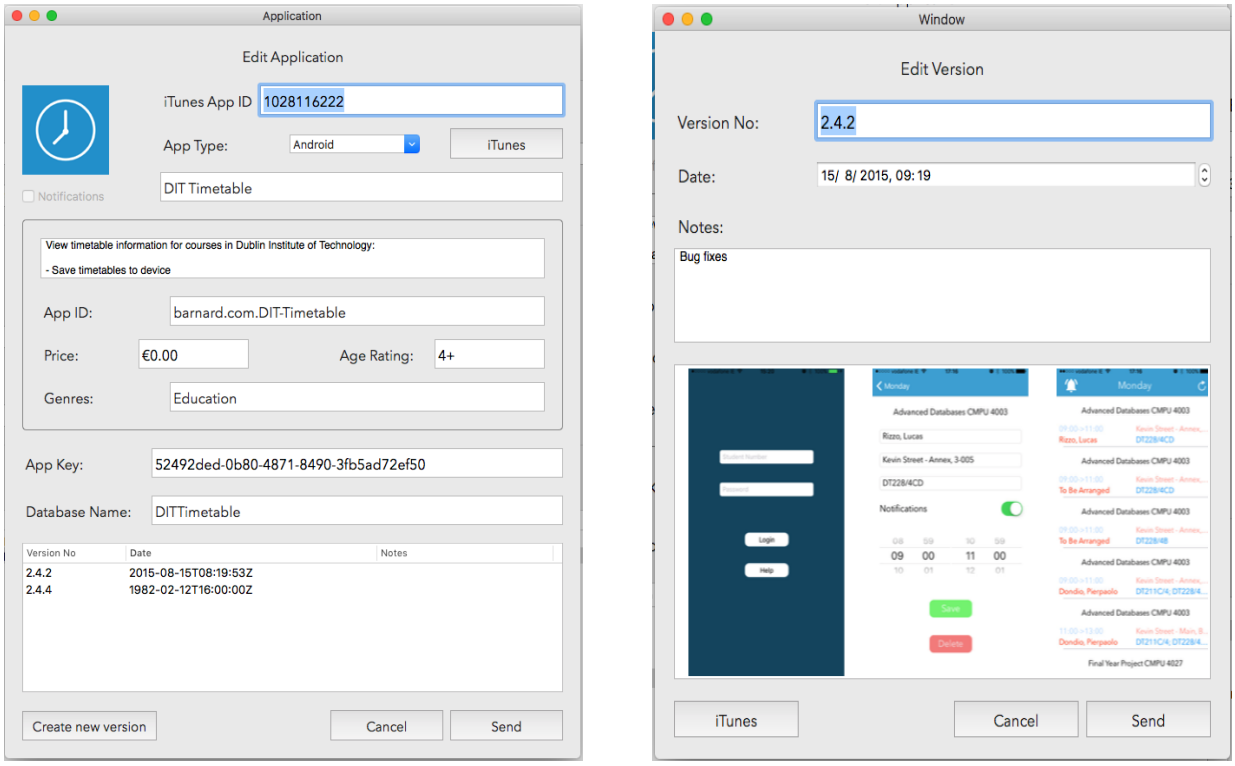


Figure 5.18 illustrates the settings view, where the configuration for the web-server and the mobile applications occurs. The view is split up into three sections; the first section has some configurations values for accessing the web-server and database. The secret key allows the mobile applications to access the web-server so that the key will be put in the header. The database username and password can be set and allows for some extra security. The dashboard includes two API request to DigitalOcean and Github, the tokens and key names in

this view are stored in *UserDefaults* to be accessed quickly when needed.

The notification section is for configuring the Apple Push Notifications (APNs), this has changed in the past year as was mentioned in the design chapter. The process now has sped up to set up the APNs on both the Apple developer console and the server. Now one key file with extension .p8 is all that is required to send APNs, this and three other values. Two of them are the team id, which is the developer's id found in the developer console, and the key id which is provided when requesting a new .p8 key file. The Select Key buttons bring up file window, to get the file from the developer's computer, and uploads the file via HTTP to the server. Once the Save button has been selected, the two values and the key file are sent to the database. The other app id value will be set when an application is being made discussed next.



(a) App View (b) App Version

Figure 5.20: Configuring Apps

Figure 5.19 illustrates the classes for the application and version and Figure 5.20 shows where these values are displayed. Each application contains several properties, and some of these are provided by Apples iTunes API. In Figure 5.20a, the button iTunes, once the user has entered the iTunes App ID field, retrieves some values which are displayed inside the box along with the app icon. The URL for making the request is *https://itunes.apple.com/lookup?id=*, and the app id is passed into the GET request, and JSON objects are returned to be parsed into objects as seen in Figure 5.21. After being parsed into iTunes object, these are then set into the TBApplication class to be saved in the database.

This can only be done once the application has been published, but the other fields can be entered until then.

Figure 5.21: iTunes API

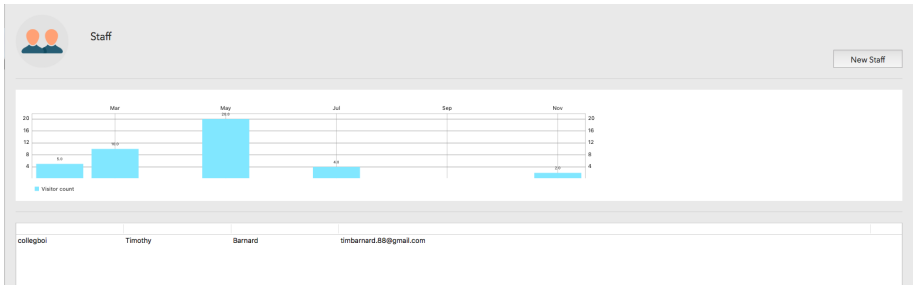

```
    ],
    "fileSizeBytes": "23177216",
    "sellerUrl": "http://www.dit.ie",
    "trackViewUrl": "https://itunes.apple.com/us/app/dit-timetable/id1028116222?mt=8&uo=4",
    "trackContentRating": "4+",
    "releaseNotes": "Bug fixes",
    "formattedPrice": "Free",
    "currency": "USD",
    "wrapperType": "software",
    "version": "2.4.2",
    "currentVersionReleaseDate": "2017-02-24T17:21:42Z",
    "artistId": 1023011697,
    "artistName": "Timothy Barnard",
    "genres": [
      "Education",
      "Utilities"
    ],
    "price": 0,
    "description": "View timetable information for courses in Dublin Institute of Technology:\n\n- Save timetables to device\n- Receive notifications before lectures/ labs begin\n- Customise timetable by
        deleting lectures/ lab\n\nDeveloped by Timothy Barnard & Stephen Fox.",
    "trackId": 1028116222,
    "trackName": "DIT Timetable",
    "bundleId": "barnard.com.DIT-Timetable",
    "primaryGenreName": "Education",
    "isVppDeviceBasedLicensingEnabled": true,
    "releaseDate": "2015-08-15T08:19:53Z",
    "minimumOsVersion": "9.0",
    "primaryGenreId": 6017,
    "sellerName": "Timothy Barnard",
    "genreIds": [
      "6017",
      "6002"
    ]
  }
 ]
}
```


The two relevant fields from this view are app key and database name, these are a security feature. Each application gets their database, by doing this keeps the data separate from other applications. The second security feature is the app key; this provides access to the database of the mobile app. As mentioned in the settings section, the secret key is sent up each request; this is also the same with the app key. This was discussed in the server section earlier.

In Figure 5.20b, the developer can keep a history of app versions published. The iTunes API already mentioned does not provide history of published version, so by having the feature gives the developer a history of what changes has been made in each version. This view also displays any notes and app stores images that have been added in that version. The iTunes button at the bottom left does the same as in Figure 5.20a, but this view retrieves different fields to be saved. The reason for this button is that when a new version is published, the app id will return the new app versions data only.

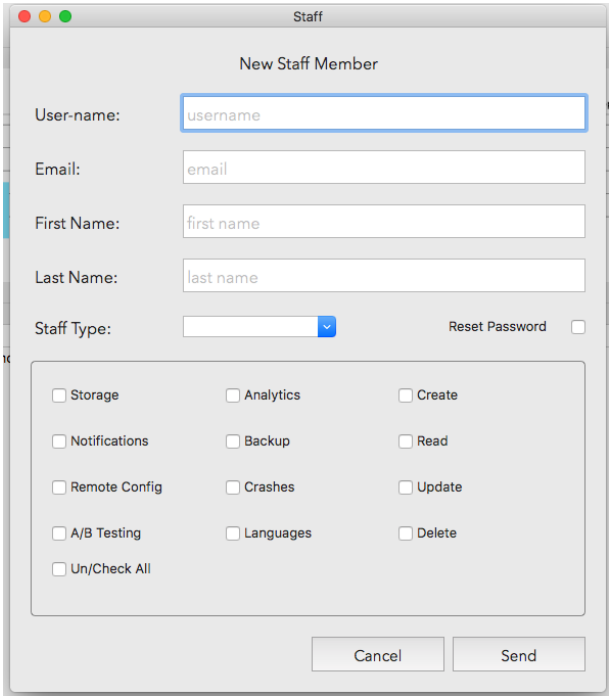
Staff

Figure 5.22: Staff View



The staff view displays all current staff that have been registered as can be seen in Figure 5.22. The bar chart at the top shows the analytics of staff members logging in at a month to month basis. This view allows staff members to be added the *New Staff* button, or edit the current user by clicking the record in the table. Once click, the *Staff Member* window as illustrated in Figure 5.23 is displayed where some configurations are to be

Figure 5.23: Edit Staff View



entered. If the current staff member is of type admin, then the entire window is enabled. The typical details such as username and email address are asked to be entered, but the check boxes inside the box allow the different user rights. The first two columns can be unchecked to limit the user of what is displayed in the side menu. The reset password if the admin is checked for a staff member if they forgot their password.

Storage

The tables illustrated in Figure 5.24 display the database contents. The first table contains the list of databases; these are the names that are set out when setting up an application in the settings view. Next once the database name of been chosen, the list of contained collections. The table on the right outputs the contents of the collection, in a parent-child format. So when a cell is clicked, if that parent has children, then it can be expanded to show the values.

The class diagram for the storage view can be seen in Figure 5.25, where both classes again conform to the TBJSONSerializable protocol. This class structure is designed to accommodate for any collection of documents retrieved. The *GenericCollection* is the first top class of the collection of records which are contained in the row variable. The row variable is of type Document array, where each Document properties are as follows: *hasChildren* for checking if that parent has children to be able to expand the row, key and value hold the values in each collection document and last the children which contain the list of children as illustrated in Figure 5.24. The *TBAnalytics* collection contains three documents; in one there is key called *tags* which also holds children for example *timeStamp*.

Figure 5.24: Storage View

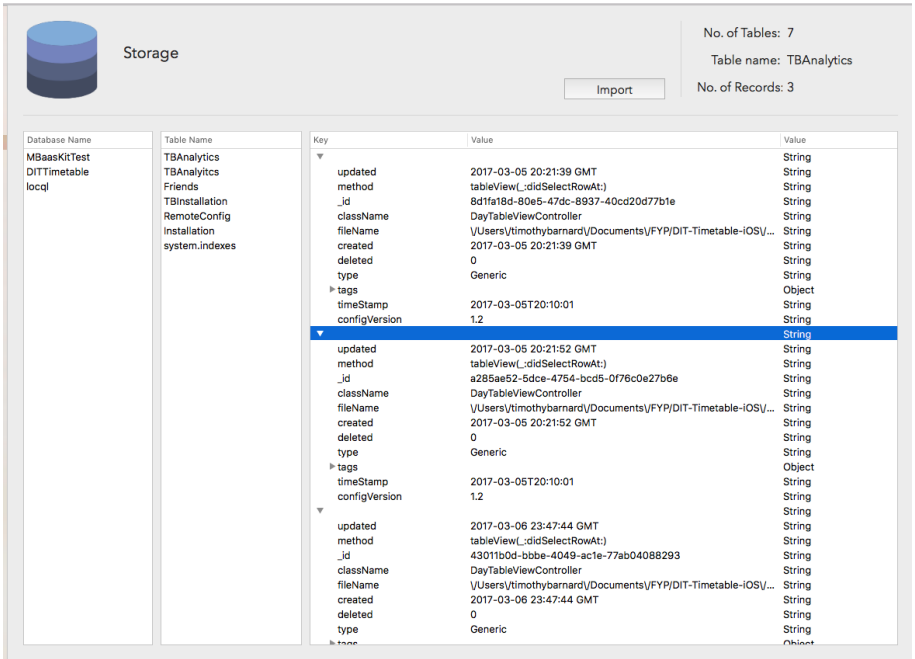
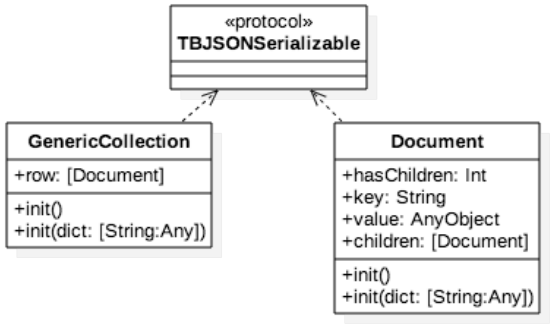
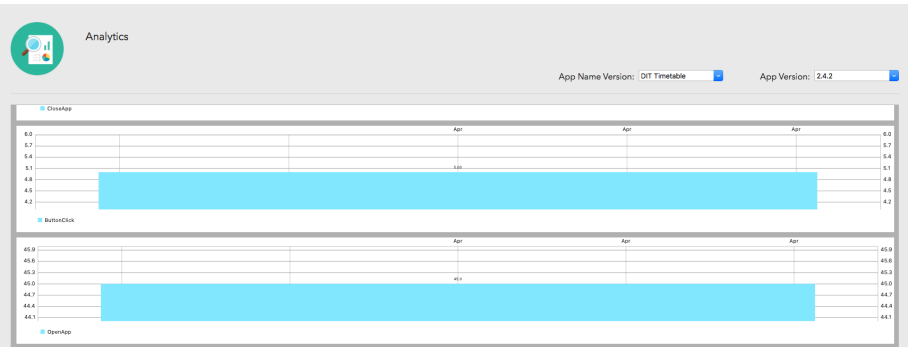


Figure 5.25: Storage class diagram



Analytics

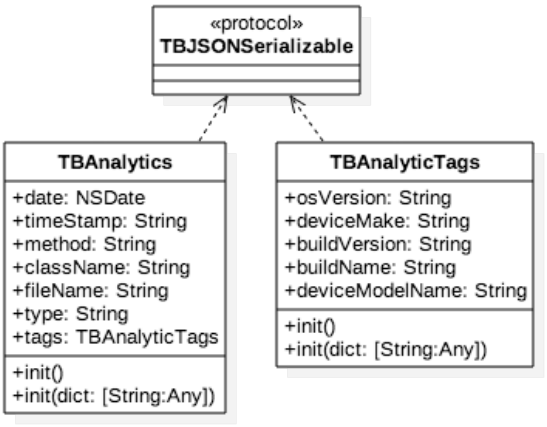
Figure 5.26: Analytics View



Illustrated in Figure 5.26, it displays the analytics gathered from the different applications and versions. The drop down lists allows the developer to change where analytics was gathered from. The scroll view below

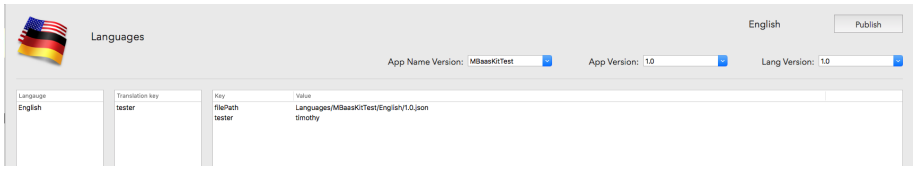
contains a collection of views which display the analytics in a bar chart form. Each bar graph view is for the different types of analytic gathering, as the analytic class diagram can be seen in figure 5.27.

Figure 5.27: Analytics class diagram



Languages

Figure 5.28: Languages View



The languages view in Figure 5.28 illustrates the layout, where the language can be selected, along with creating/updating a translation. After the mobile app and version have been chosen, the languages and translations will be displayed in the tables below. The first table shows the current languages which can be changed to active/inactive as required, which can limit if a language is an option. The next table is a collection of keys (translation keys) that is available, by separating this enables the developer to see what other translations have. This will remove any discrepancys between the translations files, e.g., between English and French.

Figure 5.29 shows the class diagram which is used in the Language View. The languages class holds properties including the name, for example, English, and if that language is available in the currently selected version of the app. The language version class is where the meta-data regarding each version of translation are kept. These include the app version, the file path to the location where the language file is being stored, and if this version is published.

Figure 5.29: Language class diagram

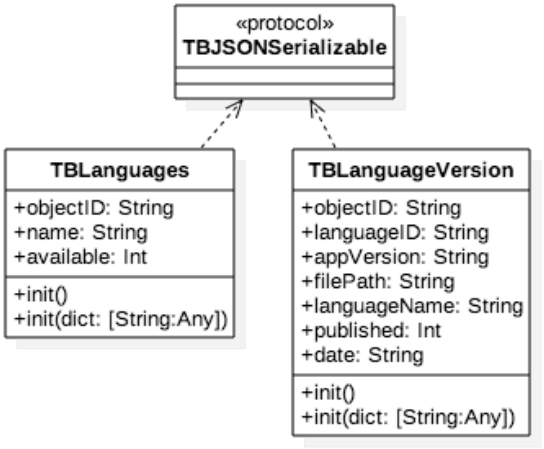
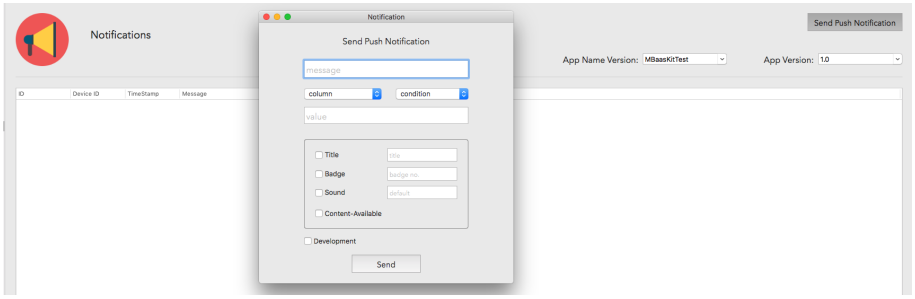


Figure 5.30: Storage View



Notifications

The notification view in Figure 5.30 has two functionalities, one for view all sent notifications, and the other two send one manually. This can be done by selecting Send Push Notification button at the top right, which will bring up a new window. The fields that can be entered are what are used when sending notifications, and the two main ones are the message itself and the unique device id. The window also supports the capability of sending multiple notifications to all devices or grouped by device type, etc. The fields inside the box are optional, the title of the notification and if not set is the application name. Once the send button is pressed, a request is made to the web server to send the notifications now.

Remote Configuration

The remote configuration view as illustrated in Figure 5.31 contains four tables which in turn represents the four classes already discussed in the services section in Figure 5.4. The figure displays the current configuration for the DIT-Timetable app with version 2.4.2. When the page initially loads, the first of each drop down list is chosen to show that current configuration. The first of the drop down menus display the list of applications, which are taken from TBApplication collection. This collection is set from the settings to view already discussed,

Figure 5.31: Remote Configuration View

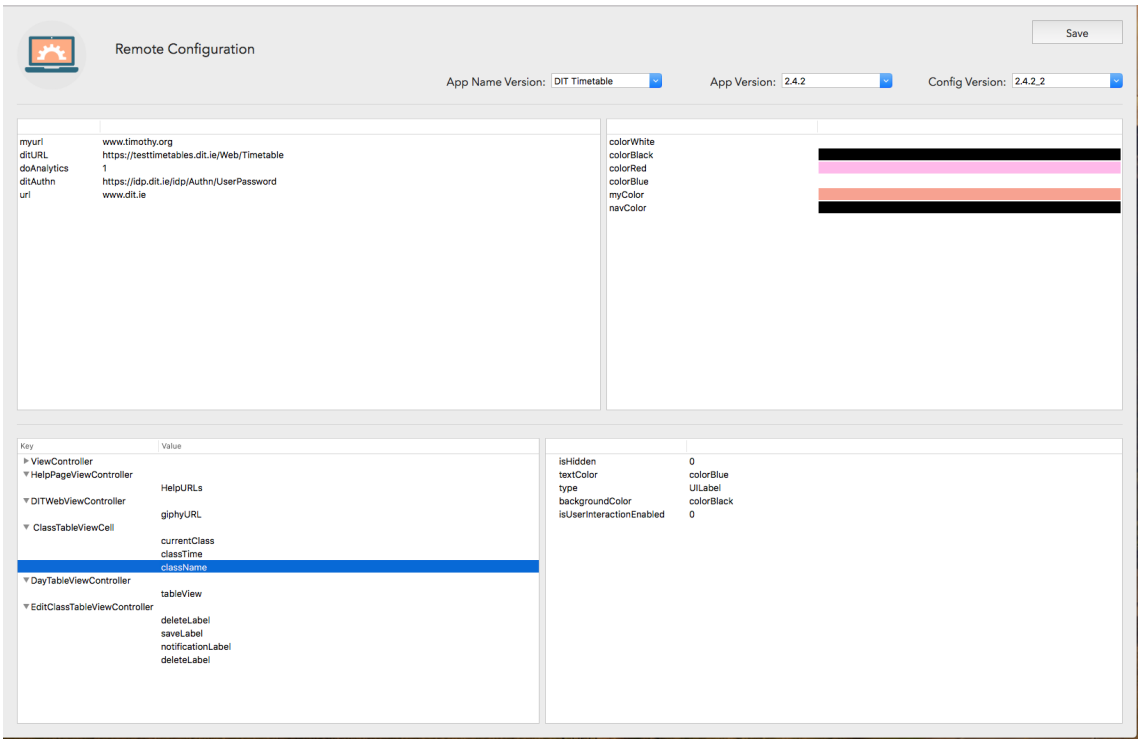
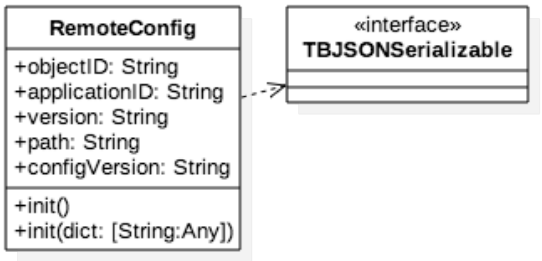


Figure 5.32: Remote Configuration View



along with the application versions, which is the contents of the next drop down. The last list contains the current configuration versions as seen in Figure 5.32. Each remote configuration object has an application and version, its relates to. The configVersion property allows for A/B Testing service which is discussed next.

Starting from the top left table which contains the main settings values, next on the right at the colours used in that particular version of the app. The bottom left table contains the view controller or classes, and if a class contains objects, the cell can be expanded to display all the objects. Once an object has been selected, the table on the right provides all possible properties that can be used. This part of the dashboard also contains a JSON file, that contains all UI objects that can select with their properties as illustrated in Listing 5.19. If a property options are a list type, then the raw string values are shown, for example, textAlignment. In Swift, the UI object property options are the type of enumeration, so when the user chooses an option, the integer value is stored. When a property has been selected, either two of views will show as illustrated in Figure 5.33.

The value can then selected and set.

```
{
  "UILabel": {
        "type": "UILabel",
        "text": "text",
        "textAlignment": [ "left","center", "right", "justified","natural"],
        "backgroundColor": "color",
        "font": "text",
        "fontSize": "number",
        "textColor": "color",
        "isEnabled": ["False", "True"],
        "isHidden": ["False", "True"],
        "isUserInteractionEnabled": ["False", "True"]
    }
}
```

Listing 5.19: UI Object JSON

Figure 5.33: Edit Property View

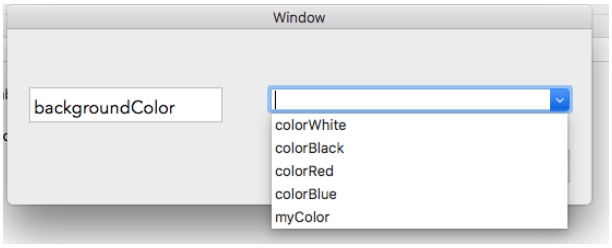
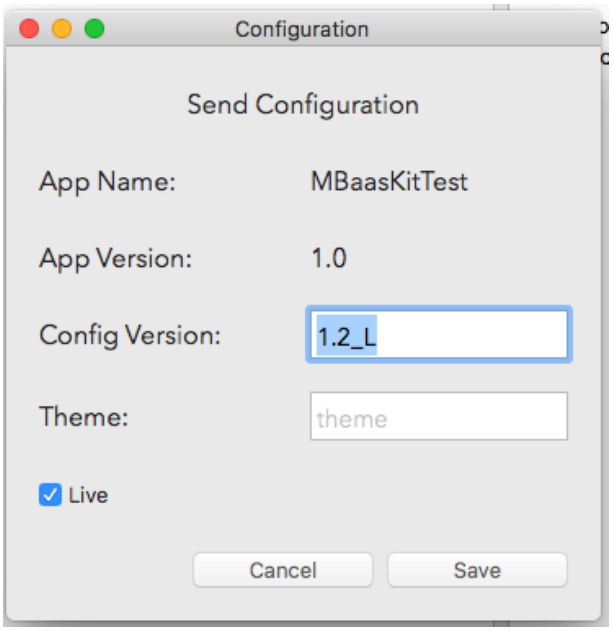


Figure 5.34: Save Configuration View



Once the configuration has been set, the user has the option to save and publish that version. This can be done

by pressing the Save button at the top right which displays a new window as illustrated in Figure 5.34. This view allows the user to set the version name and theme. The testing chapter will demonstrate the configuration of the theme in use. The live check box if unchecked can restrict this version for the mobile application to use.

AB Testing

Figure 5.35: AB Testing View

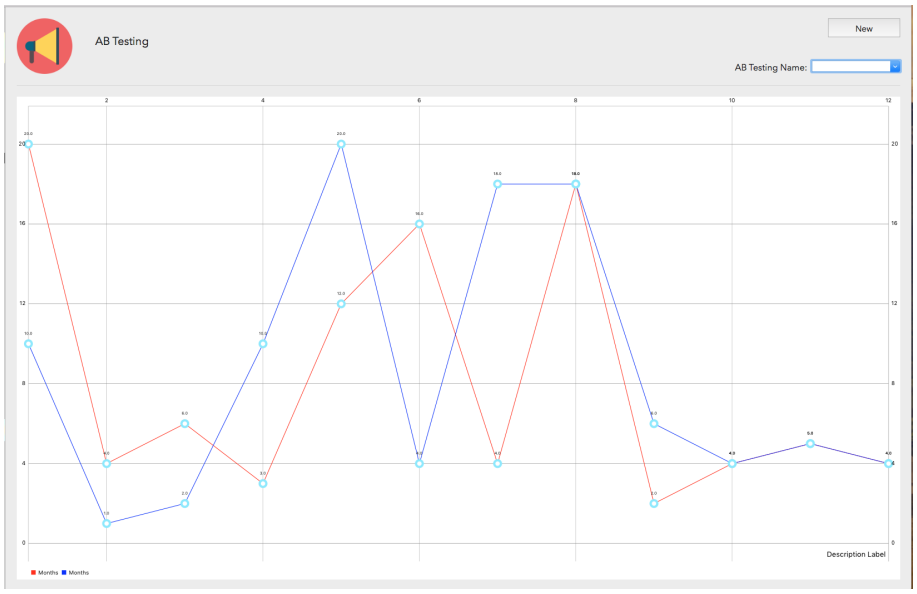


Figure 5.36: AB Testing Config View

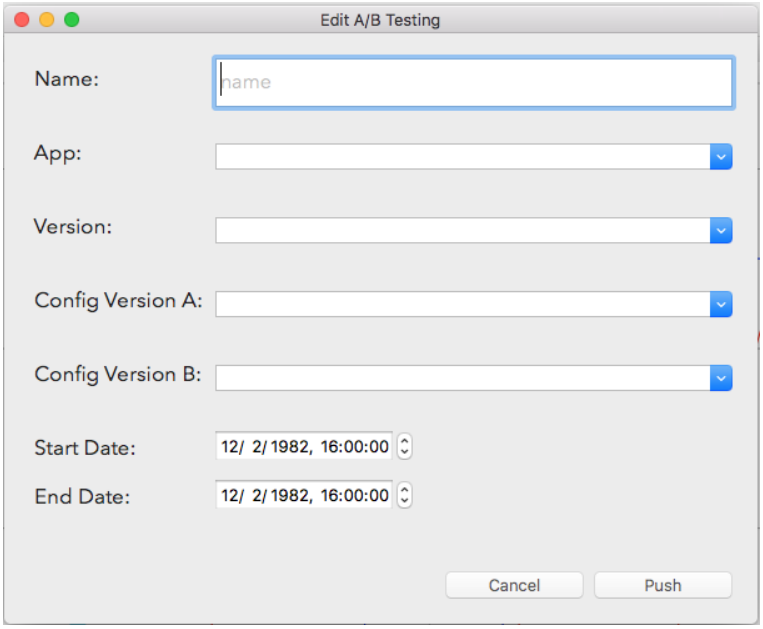


Figure 5.35 illustrates the AB Testing view, and figure 5.36 shows how to configure the A/B testing object. The main AB testing view displays a line chart with two line. Each line corresponds to a version that has been included in a particular testing set up, as was already discussed with Figure 5.5. The analytical data gathered

is from the *TBAnalytics* class that was examined in section analytics.

The drop-down list can be used to select a particular testing, and view the results to see what configuration version had the highest usage. To configure a new A/B testing, the new button is pressed to display the new in Figure 5.36. The following values are required to be set, the name, the application name, the particular version of the app and the next two drop down list are the different versions that were already configured in Remote Configuration view. The start and end time are set to allow a period for which these tests are run. Once all the values have been entered, the push button will send the new object to the database, and when a request is made to the server for a configuration file, the A/B testing object will be retrieved, and one of the versions will be shown.

Backup

Figure 5.37: Backup View

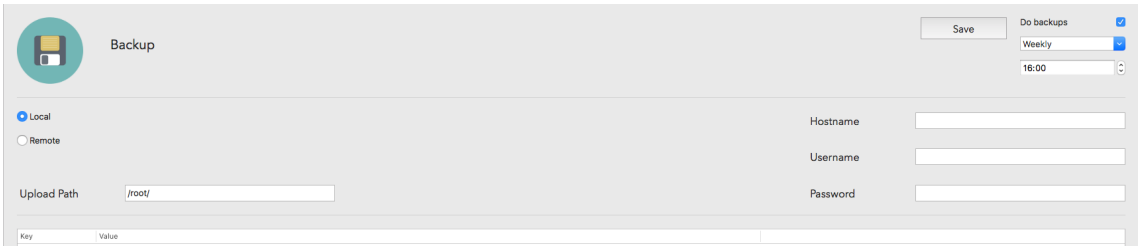


Figure 5.37 illustrates the backup view, where completed backup of the database and files can be configured. This view offers a list of configurable values, including if backups are to be done in the first place. Once checked, the drop down list can be set to daily, weekly or a particular day of the week. Next, the time of the backup can be configured; then the developer has two options to save the backups locally or remotely. If locally the folder path only needs to be entered.

Each backup folder name will contain the timetamp of when the backup occurred. After the configuration has to be done, the save button will send the values to the server and start the process. In the web-server section later, the backup web app will be discussed in detail of what happens. The table to the bottom will contain a list of backups that have been done, and by right clicking a record and clicking Delete, both the record and the backup directory will be deleted.

Issues

The issue view as illustrated in Figure 5.38 is where all exceptions from the mobile applications are displayed. The line chart shows the number of exceptions on a 12-month basis. This can give the developer a quick overview of how the application is doing. The exception class structure was already discussed in the services section. If

Figure 5.38: Issues View

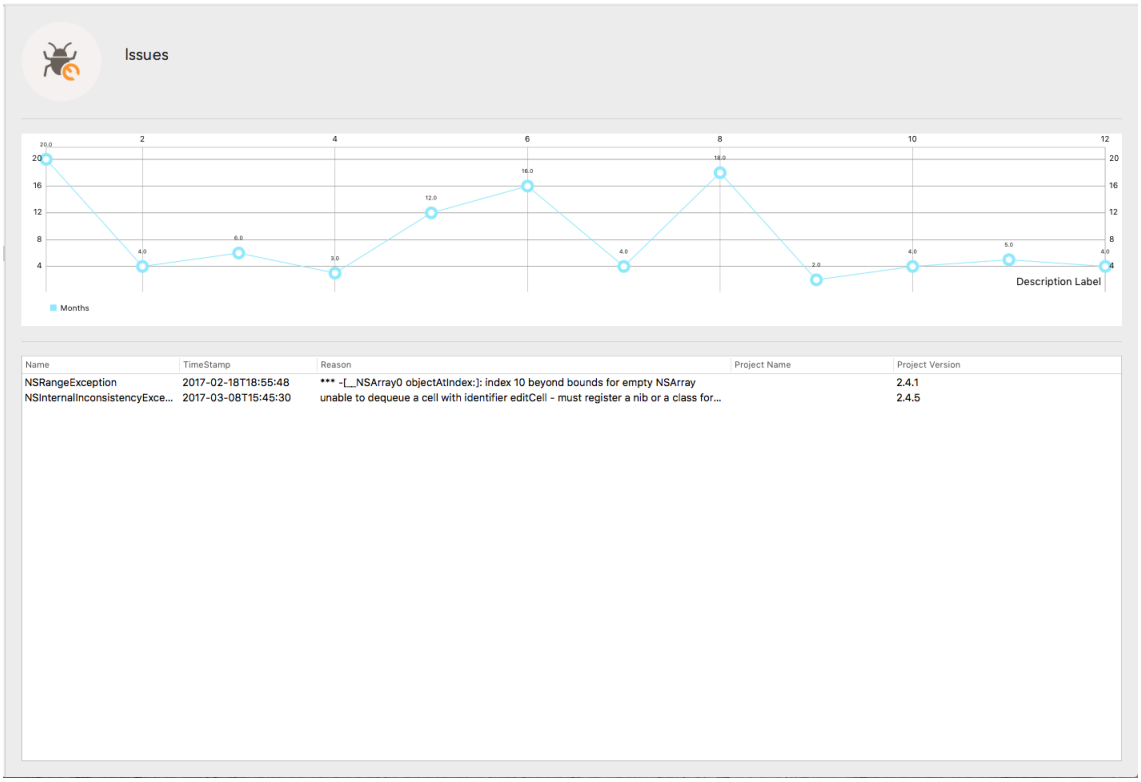
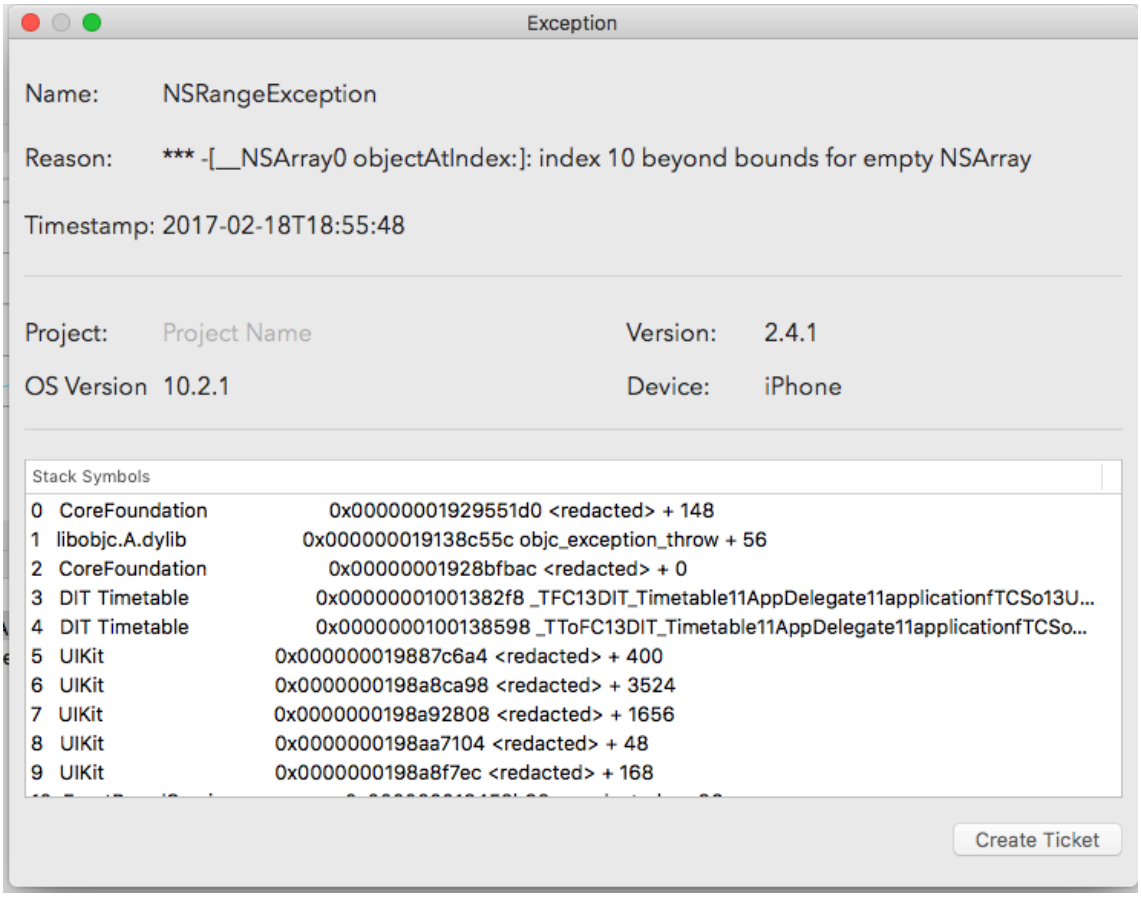


Figure 5.39: View Detailed Issue



the table below displays any exceptions, the record can be clicked to show the exception in more detail as can be seen in Figure 5.39. In this window, the issue can be attached to a new ticket which is discussed next.

Tickets

Figure 5.40: Tickets View

Figure 5.41: Issues Class Diagram

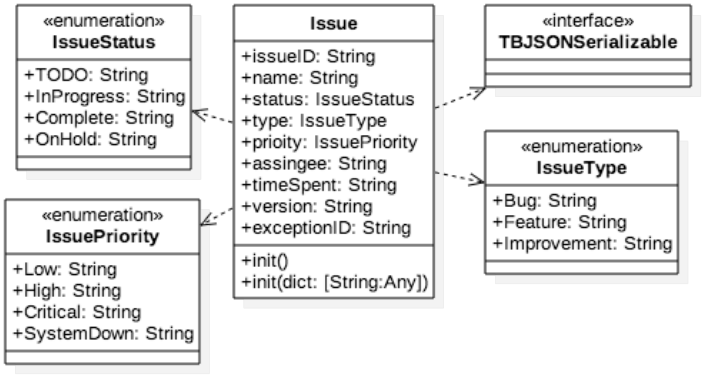


Figure 5.40 illustrates the tickets view containing all the different type of tickets being, bugs, features, etc. are displayed. After selecting the application name and version from the drop-down list, the table below will get populated with the current issues relating to that application. The class diagram for each issue is illustrated in Figure 5.41, where the Issue class has the same protocol again. This gives the class functionality to save and retrieve all the issues. The issue class consists of few enumeration type variables, where an issue can be a bug, the issue has a status and priority. An enumeration was used to make sure of the value being parsed into the database, being only one of the values.

To be able to create a new and edit an issue, a new window was developed. This view as seen in Figure 5.42 contains several values, starting off with the name of the ticket, and the assignee. The issue displayed has a type of feature, the status of *TODO* and low priority and the app version. The *"Add your comments"* section contain the logged comments regarding this ticket, which can be done several times. Each recorded comment will include the assignee and the timestamp.

Figure 5.42: Tickets View

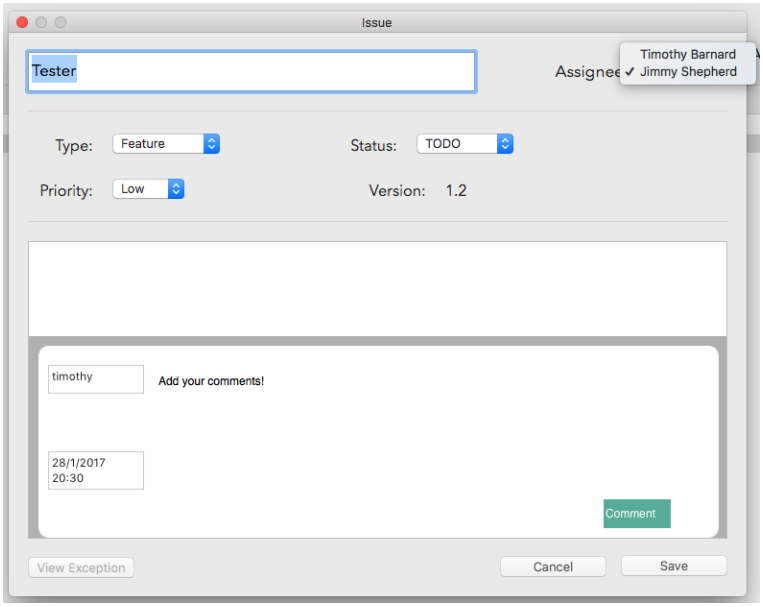
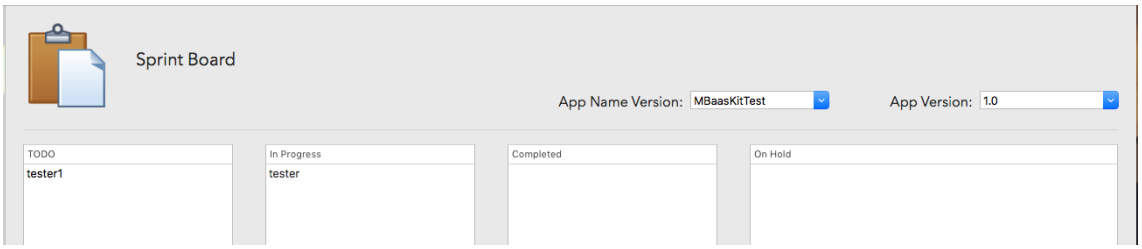


Figure 5.43: Sprint Board View



Sprint Board

Figure 5.43 illustrates the sprint board view, where the tickets/issues from the previous section will show. This view gives the developer an easier layout to monitor when working through assigned tickets. The issues can also be dragged and moved into a different table, so moving an issue from in progress section to completed. The main function for this can be seen in Listing 5.20, which looks after moving the value into the correct table.

```
1  func tableView(_ tableView: NSTableView, acceptDrop info: NSDraggingInfo, row: Int,
       dropOperation: NSTableViewDropOperation) -> Bool {
2      let data:Data = info.draggingPasteboard().data(forType: NSStringPboardType)!
3      let rowIndexes:IndexSet = NSKeyedUnarchiver.unarchiveObject(with: data) as! IndexSet
4
5      if ((tableView == todoListTableView) || (tableView == inProgressTableView)
6          || (tableView == completeTableView) || (tableView == onHoldTableView)) {
7          if ( (info.draggingSource() as! NSTableView) == tableView ) {
8              guard let newStr = self.removeFromTable(tableView: info.draggingSource() as!
       NSTableView, row: rowIndexes.first!) else {
9                  return false
10             }
```

```
            self.updateSameTable(tableView: tableView, row: rowIndexes.first!, value: newStr)
        }
        else {
            guard let newStr = self.removeFromTable(tableView: info.draggingSource() as!
    NSTableView, row: rowIndexes.first!) else {
                return false
            }
            let issue = self.addToTable(tableView: tableView, value: newStr)
            self.sendIssue(issue)
        }
        self.reloadAllTable()
        return true
    } else {
        return false
    }
}
```

Listing 5.20: Sprint

5.7 CocoaPod Framework

First, the CocoaPod was initialised, and this was done all on the command line as see in Listing 6.4. The first command installs CocoaPods, then once installed, the pod is created. The *pod lib lint* builds the skeleton structure and associated files, next to create the project; *pod lib creates MBaaSKit* was run which was followed by the terminal asking for some inputs.

```
gem install cocoapods

pod lib lint

pod lib create <pod name>

nano <pod name>.podspec

pod lib lint <pod name>.podspec
```

Listing 5.21: Pod Init

- What language do you want to use? - Swift
- Do you want to include demo application? - Yes
- What testing framework do you use? - None (this will be done later)

- Do you want to view based testing? - No

After the pod is initialised, the .podspec file required to be updated, so running *nano MBaaSKit.podspec* to edit the file. The following values are needed to be updated:

- s.summary - a brief summary of the pod
- s.description - a brief description of the pod
- s.homepage - Github link to the location of the pod

After the pod has been setup, the next listing 5.22 adds all the files and commits to local origin repository and pushes to master.

```
git add -- all
git commit -m "Initial Commit"
git remote add origin https://github.com/<GITHUB_USERNAME>/<pod name>.git
git push -u origin master
```

Listing 5.22: Pod Github

At this stage, the pod has been set up, and the initial Github commit has been pushed.

Once the project files and code has added, the next stage was to make the pod available known as pod tagging. This was done by running the following commands in Listing 5.23. The first command git tag "0.0.1" was run, which gives a version, then git push origin "0.0.1". After which pod spec lint MBaaSKit.podspec verifies that everything is configured correctly between where the source code is stored and the .podspec file. The output states "MBaaSKit.podspec passed validation", so the last command to push the pod can be done.

```
git tag <version number>
git push origin <version number>

pod spec lint <pod name>.podspec

pod trunk push <pod name>.podspec
```

Listing 5.23: Pod Tagging

The classes to be used in the library has already been implemented in the services section. The services have gone through design and testing, and now can be added to the Cocoapods framework. The classes and class functions had to be updated with the public keyword. This public keyword gives the mobile app access to these files, and this library is technically outside the project. This SDK is tested using another mobile app; this will be discussed in the testing chapter.

5.8 Conclusion

This chapter discussed the development for the three deliverables: Mac App, SDK and web server. The next chapter will demonstrate the implementation, on how to install and use the web server and SDK.

5.8.1 Code Stats

Table 5.14: Project Code Stats

Deliverable	Files	Code
Prototype iPad	24	1412
Test App	4	345
Server	35	2,911
Dashboard	275	28,138
SDK	36	2,433
Total	375	34,894

Chapter 6

Implementation

6.1 Installation Deployment

One of the key components of my project is the back-end web application. So the developer can host their back-end on any server of their choosing as long as the operating system is Ubuntu 16.04. The complete system is available on Github.

`https://github.com/collegboi/PerfectServer`

```
#!/bin/bash

#    name@ip_address
ssh root@100.100.100.100
```

Listing 6.1: Server Login

SSH or Secure Shell is a network protocol that provides a secure, encrypted way to communicate with your server. In Figure 6.1 is used to log in to your server. The username by default by root but we will change this next, and the next field is the IP address of your server.

```
#USERNAME = your new username you want to be called
useradd -d /home/USERNAME -m -s /bin/bash USERNAME
echo "USERNAME ALL=(ALL) NOPASSWD:ALL" >> /etc/sudoers

#USERNAME = your username you just set
passwd USERNAME
```

Listing 6.2: Setting user-name

In Figure 6.2 we are going to set your new username and password. Doing this will help secure your server by moving away from the default root user-name.

```
#pulling the server from github
git clone https://github.com/collegboi/PerfectServer

cd /PerfectServer
./install
```

Listing 6.3: Installing

Last is to pull down the server from Github and install all the necessary packages This is done by running the following commands one at a time in Figure 6.3.

6.2 Development

6.2.1 Add the SDK

```
$ cd your-project directory

#if you do not have cocoapods installed
$ sudo gem install cocoapods

$ pod init
```

Listing 6.4: Init pods

After creating if you don't have a Xcode project, you will want to install the SDK. This will be done using CocoaPods. CocoaPods is a dependency manager for Swift projects. It contains thousands of libraries that can be used in your apps. One of which is MBaasKit. Start off with stepping into your project as seen in Figure 6.4.

Next, we will add the MBaasKit pod to the file 6.5.

```
pod "MBaaSKit"
```

Listing 6.5: Pod file

Last part for adding the SDK to your project is to install the pod. This is done by running the command in Figure 6.6.

```
$ pod install
$ open your-project.xcworkspace
```

Listing 6.6: Pod install

6.2.2 Using the SDK

Once the SDK has been included in the project, the following values are required to be added in the *Info.plist* file.

1. APPKEY - from the dashboard settings view, in the application window
2. SERVERKEY - the secret key in the dashboard Settings view
3. URL - the domain name for the location of the web server

Remote Configuration

Remote Configuration provides the ability to maintain the application user interface in production. The framework contains a list of protocols for the UI objects such as *UITextField*, *UITableView*, etc. This protocol will grab the required properties for the object from locally stored configuration files. These files are download initially when the application is first opened, and the end user has the option depending on the developers choice of themes if any. The steps required to implement this is shown in the following Listing 6.7.

```
1 @IBOutlet weak var tfName: RCTextField!
2
3 self.tfName.setupLabelView(className: self, name: "tfName")
```

Listing 6.7: Remote Configuration Setup

In listing 6.8 illustrates how downloading a new theme that the end user has selected. The call back function takes the parameter of the theme name, once the call back has been called, updating the view can be done. Line 4 is updating the configuration file with the reading file name. This can be changed to be included in a function that handles when the app goes in the background depending on choice. The *updateViews()* function can be implemented to refresh the view if changes are immediate.

```
1 RCConfigManager.getConfigThemeVersion(theme: theme) { ( complete, message) in
2     DispatchQueue.main.async {
3         if complete {
4             RCConfigManager.updateConfigFileNames(fileType: .config)
5             self.updateViews()
6             self.showMessage(title: "Theme", message: theme + " successfully downloaded")
7         }
8     }
9 }
```

Listing 6.8: Retrieving new theme

Notifications

To start using notifications, we first need to create the installation object and send that to our server. This is done by adding the following function in Listing 6.10 in the *AppDelegate* file.

```
func application(_ application: UIApplication,
    didRegisterForRemoteNotificationsWithDeviceToken deviceToken: Data) {
  let installation = Installation(deviceToken: deviceToken)
  installation.sendInBackground() { ( completed, message) in
    DispatchQueue.main.async {
      if (completed) {
        print(message)
      }
    }
  }
}
```

Listing 6.9: Register for Notifications

The sending of the notification is illustrated in Listing 6.10. The values being set are the title, the receiver's user id and the message.

```
  let newNotification = TBNotification()
    newNotification.setTitle("test")
    newNotification.setUserID(11)
    newNotification.setMessage("hello")
    newNotification.sendNotification { (sent, message) in
        DispatchQueue.main.async {
            if sent {
                print("notification sent")
            }
        }
    }
```

Listing 6.10: Send Notifications

Storage

```
struct TestObject: TBSONSerializable {

  var name: String!
  init() {}
  init(name:String) {
    self.name = name
  }
```

```
8   init( dict: [String:Any] ) {
9     self.name = dict["name"] as! String
10  }
11 }
12
13 var result = [TestObject]()
14 result.getAllInBackground(ofType:TestObject.self) { (succeeded: Bool, data: [TestObject]) ->
       () in
15   DispatchQueue.main.async {
16     if (succeeded) {
17       result = data
18       print("success")
19     } else {
20       print("error")
21     }
22   }
23 }
24
25 let testObject = TestObject(name: "timothy")
26 // if objectID is set then it is updating else inserting
27 testObject.sendInBackground("<objectID>"){ (succeeded: Bool, data: NSData) -> () in
28   DispatchQueue.main.async {
29     if (succeeded) {
30       print("scucess")
31     } else {
32       print("error")
33     }
34   }
35 }
```

Listing 6.11: Storing/Retrieving Objects

As can be seen in Figure 6.11, when creating a struct and using the protocol *TBJSONSerialiszable*, we can then send and retrieve the objects in the backgrounds using those commands.

Exception Handling

An exception is a problem that arises during the execution of a program. If exceptions are not handled, it can cause the app to crashes, and when the app is live, there is no way of knowing. But by adding the following lines 1 and 2 in Listing 6.12 the uncaught exceptions can be caught and sent to the back-end storage and view with the dashboard. Line 4 is an example of sending a caught exception that can have a message sent along.

```
1 MyException.client()
2 MyException.sharedClient?.setupExceptionHandler()
```

```
3
4 MyException.sharedClient?.captureMessage(message: "Caught Exception")
```

Listing 6.12: Storing/Retrieving Objects

Chapter 7

Testing and Evaluation

7.1 Testing

Software testing is an investigation conducted to provide information about the quality of the product or service under test. Testing is executing a system to identify any gaps, errors, or missing requirements in contrary to the actual conditions. In this project the test-driven approach method was used, this helps eliminate parts of the code that will not work or fix them before adding them to the system.

7.1.1 Service testing

As this project is Test Driven Development approach, all service unit testing has previously been done. These tests were done by building a small client application in Playground and web-server in Perfect which was developed and ran in Xcode. The API requests were tested afterwards using the Postman application, to run each API request and ensure the result is as expected. The tests included checking the secret key and application key were authenticated, and if an incorrect key were used, then an error would be returned. In Table 7.1 are some of the test requests made along with the response, where the second test has failed due to missing authentication keys.

Table 7.1: Service Testing

API Call	HTTP method	Response	Body	Result
http://perfectserver.site/api/cc2b14fa-f59b-4e13-905a-3eebaf2ff659/storage/RemoteConfig	GET	Figure 7.1		PASS
http://perfectserver.site/storage/TBAnalyitcs/	GET	{"result": "error" ,"message": "access rights"}		FAIL
http://perfectserver.site/api/cc2b14fa-f59b-4e13-905a-3eebaf2ff659/storage/Friends	GET	Figure 7.2		PASS
http://perfectserver.site/api/cc2b14fa-f59b-4e13-905a-3eebaf2ff659/storage/Friends	POST	{"result": "success", "message": "0e9635f6-b3fd-408d-b4a2-458dab34c781"}	{"dob": "12/12/2016", "age": "44", "name": "Jimmy" ,"county": "Portsmouth" ,"country": "England"}	PASS

Figure 7.1: API Call 1


```
{
  "config": {
    "version": "0.0",
    "date": "0.0"
  },
  "count": 3,
  "data": [
    {
      "_id": "f6cd907a-61ae-4264-b664-55fc1aa15fa3",
      "version": "1.0",
      "path": "ConfigFiles/MBaasKitTest/config_1.2_L.json",
      "applicationID": "29c82223-96d0-47c5-8a78-af0994aaeeda",
      "appTheme": "Dark",
```


Figure 7.2: API Call 2


```
{
  "config": {
    "version": "0.0",
    "date": "0.0"
  },
  "count": 7,
  "data": [
    {
      "_id": "15183857-496c-439f-9435-5fd8bb1432b9",
      "dob": "12/12/2016",
      "age": "21",
      "name": "Tim",
      "county": "Dublin",
      "country": "Ireland ",
      "created": "2017-03-05 18:34:43 GMT",
      "updated": "2017-03-05 18:34:43 GMT",
      "deleted": "0"
    },
```


7.1.2 Integrated App Testing

This part of testing involved developing a new mobile application with a simple user interface. The SDK developed was downloaded uses Cocoapods and installed in the testing app workspace. The application illustrated in the following Figures 7.4a and 7.4b, displays a table view where an array of Friend are shown. The objective of creating this test application is to demonstrate the remote configuration that allows the user to choose a new

theme in Figure 7.3, and instantly see the difference.

The project aim from the beginning was to speed up development, testing and when the app is live. The Figures 7.4a and 7.4b illustrates how the app can be updated when the application is live quickly. The next three listings demonstrate on what is required in the development stage to be able to refresh the UI objects when the app is published.

Listing 7.1 is UI objects include label and button, and these in one line can be initialised to start using the remote configuration feature.

```
self.tfName.setupLabelView(className: self, name: "tfName")
self.tfAge.setupLabelView(className: self, name: "tfAge")
self.tfDOB.setupLabelView(className: self, name: "tfDOB")
self.tfCountry.setupLabelView(className: self, name: "tfCountry")
self.tfCounty.setupLabelView(className: self, name: "tfCounty")
self.sendFriend.setupButton(className: self, "sendFriend")
self.sendAlert.setupButton(className: self, "sendAlert")
```

Listing 7.1: Remote Config demo1

Listing 7.2 illustrates how the remote configuration file is being retrieved based on the theme chosen.

```
func getDiffLanguage( theme: String ) {
    RCConfigManager.getConfigThemeVersion(theme: theme) { ( complete, message) in
        DispatchQueue.main.async {
            if complete {
                print(theme)
                self.updateViews()
                self.showMessage(title: "Theme", message: theme + " successfully downloaded")
            }
        }
    }
}
```

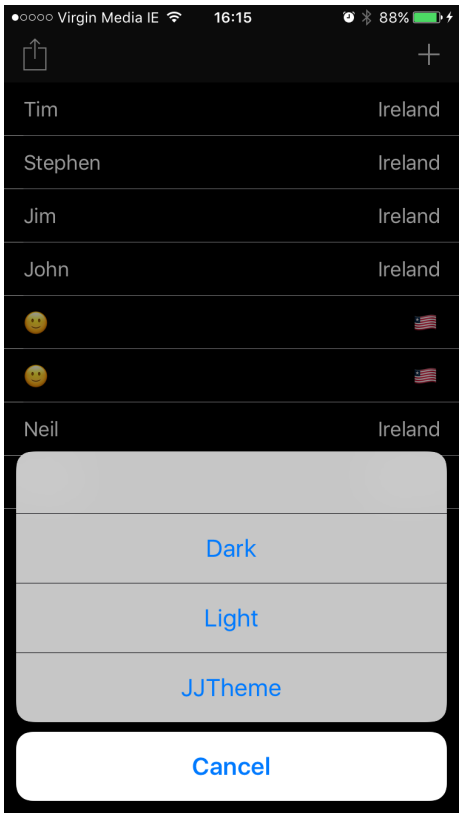
Listing 7.2: Remote Config demo2

Listing 7.2, the configuration file is being updated. This means the old file is being deleted, and the new file updated to the current name and location. Next, the navigation bar is being updated.

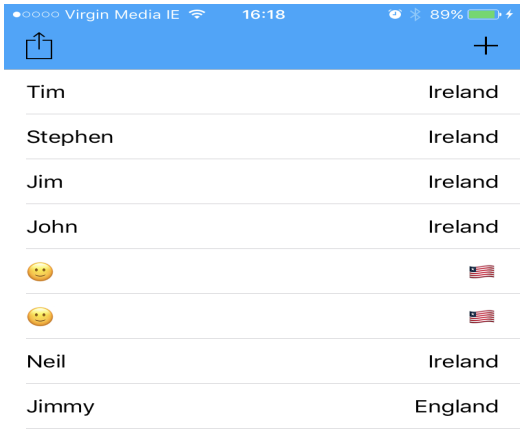
```
RCConfigManager.updateConfigFileNames(fileType: .config)
RCConfigManager.updateNavigationBar(className: self,objectName: "navBar");
```

Listing 7.3: Remote Config demo3

Figure 7.3: App Theme options



(a) Dark Theme



(b) App Version

Figure 7.4: App Themes

7.1.3 Sanity testing

Sanity testing is kind of software testing which is performed once a build is completed with changes made. The goal is to run through the functionality of the application, and ensure they work as expected. Sanity testing is also carried out to check that the reported bugs fixed and new features do not break previously working functionality. The kind of testing is performed by testers which were done using college colleagues. A list of tests was given as seen in Table 7.2 along with the average result found. The Table 7.2 tests refer to the application stated above in integrated app testing section.

Table 7.2: Sanity testing 1

No	Description	Initial state	Steps	Expected Result	Result
1	Add new Friend	App opened	1. Selected add option 2. Fill in details 3. Press send	Friend has been added and view returns to list of friends view	Pass
2	Update Friend	Friends list view	1. Select friend to update 2. Update values 3. Press send	Friend is updated and view returns to list of friends with friend updated	Pass
3	Remove Friend	Friends list view	1. Slide cell with friend 2. Press delete	Friend is deleted, and record is gone	Pass
4	Change Theme	App opened	1. Select icon at top left 2. Choose theme ie. Dark	A pop up message stating Dark Them downloaded, and UI view is changed accordingly	Pass
5	Change Language	App opened	1. Select icon at top left 2. Choose language ie. English	A pop up messaging stating English downloaded, and title bar caption is changed	Pass

7.2 Evaluation

Throughout the project, continuous communication with the outsourced developers from Trust5 and Tapadoo was done to ensure the changes made was acceptable. The substantial changes made were moving from Django web server to Perfect (server side swift), and the dashboard from iPad to Mac app. Once the project was completed, an evaluation was done by two developers from Trust5. The first developer gave me a non-technical feedback, while the second did. The full feedback report is in appendices chapter under the evaluation section A.2.

The general feedback is good based on the usability of the library, and how effortless using the library is, developer 2 states: "the MBaaSKit library to be very easy to integrate with any iOS app". He also went on to say that using Cocoapods was a good choice as this "is the most common practice to include libraries".

One of interesting feedback that wasn't considered before was the fact that the library can be used a "functional test tool". This enables the developer to load different configuration files and see which one looks the best without the need for a new build.

The developer also offered some constructive feedback where improvements can be made, such as expanding the documentation on the Github page. He also suggests making the library compatible with Objective-C as it is still being used.

Chapter 8

Conclusion

8.1 Introduction

In this conclusion chapter, the project plan, future work, strengths and weaknesses of the project will be discussed. Reflective and learning outcomes will also be included.

8.2 Project plan

The objective plan for this project was to design and develop a way to improve mobile applications in development, testing, production and can also include acceptance (DTAP). The project has changed from the beginning within a few months into it. It started out with developing the back-end using Django framework and Python for the programming language. This changed to using Perfect (Swift server side) for the back-end and meaning that Swift was used for both client and server. The reasons behind this because the project will be open sourced, it will allow developers are already developing their apps in Swift to contribute towards it. Having software open source has its benefit which not only include having developers contributing and in return having free software. It can also help with providing more functionality, services and security to the system.

The initial plan for the dashboard was to develop it for an iPad application but changed after some thought and discussions with other developers. The dashboard was designed and developed for Mac app, and the main reason for this change was to not restrict developers from having to have an iPad. Professional developers in a company could have access to an iPad, but as this project is trying to reach new developers, then this would have been an issue.

One of the parts of the remote configuration initially was to be able to move the objects in each view, so where

a label or text field is positioned. After having a discussion with a professional mobile developer, this part was put aside. His reasoning was that constraints which are how the UI objects are positioned together are already a fully integrated set-up.

8.3 Future work

Plans for the future include redesigning the dashboard interface, due to time this could not have been done. In the services section, each view requires the user to choose the particular mobile application and version every-time the view is opened. This idea behind this was to able to stay on one screen and complete the tasks for every version necessary, and the equivalent on the other screens. It can disrupt the flow when having to keep changing the application, and also mistakes can be made when jumping through different versions.

Another plan is to get more feedback from developers as they begin to use the system, to see where the project can go. The remote configuration feature can be expanded to more UI objects within the application, and due to time, was restricted to a handful. There more features that can be added to give the developers more tools to maintain their apps. Some of these include a live database, OAuth. The live database creates a continuous connection between the end users and their data, so not having to keep refreshing a view for updates. OAuth is an extra security feature that can be included to keep usernames and password and other information private. OAuth uses other services such as Facebook, and Google to log-in and is authenticated through their services.

The next big plan is to develop a framework for Android applications, the type that was designed in this project for iOS. By doing this can broaden the scope that this system can reach and be integrated into mobile applications. The web-server is already configured to handle request from different mediums, as it uses a common protocol called an API. The remote configuration which defines the properties of the UI objects will need some configuration to add the Android framework.

8.4 Project Weaknesses

One of the weaknesses of this project, as talked about in the plans, is the design of the dashboard. The additional requirement to keep choosing the mobile application name and version from the drop-down list in some of the views. Although this has some benefits of been able to stay on one view and update multiple of apps, this could cause issues with updating the wrong version or app. Another weakness of the project but is mainly due to time, is that the system is currently limited to iOS devices, but plans are in place to changed this. This weakness is one of the largest of this project, as the statics of the highest number of mobile apps is Android, so bringing this project to that area will increase the likelihood of the system being widely used.

8.5 Project Strengths

The strength of this project is that it proved that mobile application in developed could be improved, by reducing the amount of work required. The current way applications are designed can be changed, give end users freedom of how the looks and feels with some restriction. Unconsciously people act differently on the appearance of the apps, so been able to provide them with some rights to change can in turn potentially make them want to keep and use them. The configuration for setting up the web-server, and including the SDK in the apps is developed to make easy to use. Then to use the SDK and communicate with their web-server has the added benefit of simplicity.

8.6 Learning outcomes

Personal development has been with research, the amount of done with this project out ways any other project done before. Researching my project idea has shown me skills to when developing software always to think ten steps ahead of where possible. Instead of starting the development and studying along the way but start with the research has shown a whole new way to creating software. This way of building software has advantages such as finding the potential issues and risks early on to overcome. Speaking with professional mobile developers when I did the research has helped with the project with not only validating my idea that it has potential but also giving me constructive feedback. The feedback provided has made me do more research regarding other services that companies are doing to look into. One developer I had an interview with also pointed out some areas to be cautious about. The research gone into this project apart from technologies and methodologies being used has shown the need for a mobile backend as a service.

8.7 Conclusion

The primary aim of this project was to create a system that can bring more people to start developing mobile applications. To bring a new way of developing an application, and maintaining the application quicker with ease. After getting the project evaluated from experienced developers leading to a conclusion that this project has plausibility.

Bibliography

[1] Ted Schadler. Ted schadler's blog. Available at http://blogs.forrester.com/ted_schadler/13-11-20-mobile_needs_a_four_tier_engagement_platform?utm_source=time-to-move-to-a-four-tier-application-architecture&utm_medium=blog, (Accessed Date November 2016).

[2] Backendless. Available at https://backendless.com/what-is-backend-as-a-service/, (Accessed Date October 2016).

[3] Apns. Available at https://loopback.io/images/9830524.png, (Accessed Date January 2017).

[4] Apple. Available at https://developer.apple.com/library/content/documentation/MacOSX/Conceptual/OSX_Technology_Overview/CocoaApplicationLayer/CocoaApplicationLayer.html, (Accessed Date February 2017).

[5] Kinvery. Backend as a service whitepaper. Available at http://go.kinvey.com/baas-ecosystem-whitepaper/, (Accessed Date March 2017).

[6] Trust5. Available at http://www.trust5.com, (Accessed Date November 2016).

[7] Home — tapadoo. Available at https://tapadoo.com, (Accessed Date November 2016).

[8] Apple. About objective-c. Available at https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html, (Accessed Date October 2016).

[9] Apple. Swift - apple developer. Available at https://developer.apple.com/swift/, (Accessed Date October 2016).

[10] Deviq. Available at http://deviq.com/don-t-repeat-yourself/, [cited October 2016].

[11] Welcome to python.org. Available at https://www.python.org, (Accessed Date October 2016).

[12] Perfect. Available at http://perfect.org/, (Accessed Date January 2017).

[13] IBM. Kitura. Available at http://www.kitura.io, (Accessed Date January 2017).

[14] The web framework for perfectionists with deadlines — django. Available at https://www.djangoproject.com, (Accessed Date October 2016).

[15] Welcome — flask (a python microframework). Available at http://flask.pocoo.org, (Accessed Date October 20160.

[16] Foundation n. node.js. Available at https://nodejs.org, (Accessed Date October 2016).

[17] Coocapods. Available at https://cocoapods.org, (Accessed Date March 2017).

[18] Carthage. Available at https://github.com/Carthage/Carthage, (Accessed Date March 2017).

[19] Mysql. Available at https://www.mysql.com, (Accessed Date October 2016).

[20] mongodb— for giant ideas. Available at https://www.mongodb.com, (Accessed Date October 2016).

[21] Postgresql: The world's most advanced open source database. Available at https://www.postgresql.org, (Accessed Date October 2016).

[22] Apache. Available at https://httpd.apache.org, (Accessed Date October 2016).

[23] Nginx. Available at https://www.nginx.com/resources/wiki/, (Accessed Date October 2016).

[24] Amazon. Available at https://aws.amazon.com/ec2/?nc2=h_m1, (Accessed Date January 2017).

[25] Digitalocean. Available at https://www.digitalocean.com, (Accessed Date January 2017).

[26] Django vs flask: Which is better for your web app? Available at http://ddi-dev.com/blog/programming/django-vs-flask-which-better-your-web-app/, (Accessed Date November 2016).

[27] A-Coding. A review of ios dependency managers. Available at https://a-coding.com/a-review-of-ios-dependency-managers/, (Accessed Date March 2017).

[28] Benchmarks for the top server-side swift frameworks vs. node.js. Available at https://medium.com/@rymcol/benchmarks-for-the-top-server-side-swift-frameworks-vs-node-js-24460cfe0beb#.3cmlf3psm, (Accessed Date January 2017).

[29] Perfect github repository. Available at https://github.com/PerfectlySoft/Perfect, (Accessed Date January 2017).

[30] When to use mongodb rather than mysql (or other rdbms): The billing example - dzone database. Available at https://dzone.com/articles/when-use-mongodb-rather-mysql, (Accessed Date November 2016).

[31] Digitalocean vs aws. Available at https://www.upguard.com/articles/digitalocean-vs-aws, (Accessed Date October 2016).

[32] Nginx vs apache. Available at https://www.upguard.com/articles/apache-vs-nginx, (Accessed Date October 2016).

[33] Xcode- apple developer. Available at https://developer.apple.com/xcode/, (Accessed Date September 2016).

[34] Postman — supercharge your api workflow. Available at https://www.getpostman.com, (Accessed Date September 2016).

[35] Apns overview. Available at https://developer.apple.com/library/content/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/APNSOverview.html#//apple_ref/doc/uid/TP40008194-CH8-SW1, (Accessed Date September 2016).

[36] Github. Available at https://github.com, (Accessed Date January 2017).

[37] Perfect assistant. Available at http://perfect.org/en/assistant/, (Accessed Date January 2017).

[38] Oauth 2. Available at https://oauth.net/2/, (Accessed Date March 2017).

[39] Parse. Available at https://parse.com/, (Accessed Date October 2016).

[40] Google. Firebase — app success made simple. Available at https://firebase.google.com/, (Accessed Date September 2016).

[41] Google. Pricing — firebase. Available at https://firebase.google.com/pricing/, (Accessed Date October 2016).

[42] Baasbox build awesome apps faster. Available at http://www.baasbox.com/en/, (Accessed Date September 2016).

[43] Amazon. Pricing — amazon web services, inc. 2016. Available at https://aws.amazon.com/ec2, (Accessed Date September 2016).

[44] macos human interface guidelines. Available at https://developer.apple.com/library/content/documentation/UserExperience/Conceptual/OSXHIGuidelines/DesignPrinciples.html#//apple_ref/doc/uid/20000957-CH18-SW1, (Accessed Date February 2017).

[45] Apple. Available at https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Protocols.html, (Accessed Date February 2017).

[46] Swift. Available at https://swift.org/package-manager/#conceptual-overview, (Accessed Date March 2017).

[47] Github. Available at https://github.com/danielgindi/Charts, (Accessed Date February 2017).

[48] McCarthy, Patrick, <pmccarthy@trust5.com>, "Re: DIT Final Year Project", 11 March 2017.

[49] Hilal, Gamal, <gamal@trust5.com>, "Re: Final Year Project", 24 March 2017.

Appendix A

Appendices

A.1 Research

A.1.1 Tapadoo

Tapadoo [7] gave me more constructive feedback, stating that while the remote configuration service on its own is good it has its drawbacks. A developer's point of view is that updating the user interface will not happen often enough to validate the use of it. However using the remote configuration along with A/B testing (comparing two variations of an app against each other) is a powerful, useful tool for developers and customers. Instead of relying on what the designer or the project manager thinks, they leave it up to the users by viewing the analytics based on the two different variations.

He also mentioned to be careful when using the translation files and allowing the users to choose their own language as this goes against Googles and Apples guidelines. He stated that there are services already implemented called Localization that deals with the displaying the correct translation. He also explained a need for updating content within an app, changing the button title from Pay to Pay Now for example and that my project should include this service.

A.1.2 Trust5

Trust5 [6] had two developers to meet with me, they expressed interest in the idea and said it is a powerful tool for developers to use. They gave me feedback to design it as a white label product, meaning that the product can be used by any company with their logo attached. We discussed the different features already implemented with regards the remote configuration and explained where to concentrate on and what to leave

to last. The configuration is split up into three phases of testing as explained earlier, they talked about leaving the complicated part of converting user interface objects to Apples visual format language to last and just used the constraint object class until the majority of the project was completed.

A.2 Evaluation

A.2.1 Trust5

Developer 1

I found that the project was carried out in a professional manner and to a high standard. The concept was intriguing and well executed. Seeing the configuration changes being applied in a real-time and remote manner without need for client-side deployment is impressive. The fact that this remote configuration obviates the need for a client update to be released adds significant value as this is a time-consuming and painful process, in my experience this is particularly true for the Apple apps. I've no doubt that there are many real world scenarios that could leverage this functionality. [48]

Developer 2

Library usability:

We have found the MBaaSKit library to be very easy to integrate with any iOS app. As a pod library it involved very little setup effort as it only takes one line of code to include the library and another one line command to import the library. Using Cocoapods and pod files is the most common practice to include dependencies/libraries.

Library capabilities:

We were really impressed with the capabilities of the MBaaSKit library. It is a really powerful tool that can save a lot of development time and effort especially on the design side of things. A developer can quickly create a mockup of any native view and subsequently change the look feel of that view remotely. It can also help an app but hiding or disabling a button/feature dynamically.

Obviously, one can create an app that is essentially a HTML wrapper that displays content based on a HTML that is fetched remotely. However, that has two main disadvantages which are performance security. With MBaaSKit it could give us the flexibility of changing content dynamically and without having to do a new release every time designers come up with a tweaked design. It can also help apps that are essentially the same but branded differently to different clients.

Also the library can be used as a functional test tool. The library can help load the app with some test configurations dynamically such as long text, non-latin characters, text alignment, etc..

For MBaaSKit to be useful; it also had to be a simple tool to use. Following the Sample project; it was evident that it was very simple to use as any UI element can be configured with only one line of code.

Simplicity of the library is of utmost importance as we see the selling point of the tool, apart from providing flexibility and extending an apps capability , is reducing development time.

General feedback (Improvements):

A nice feature to have is to be able to load different configurations based on a per user (or user type) level. For example an app developer might want to show or enable different areas of a view depending on the type of the user using the app (e.g. basic vs premium user).

Improvements can be made to provide more documentation on the GitHub page on how to use the library with extra screenshots of the librarys capabilities.

Also some guidance could be provided on how to use the library with an Objective-C project.

The sample project provides great examples on how to use the library and showcases its capabilities. However, we found the name of the sample project MBaasKitTest to be a little bit confusing as we werent sure initially if it was a test or a sample project. [49]