# 

# ZaCloud Cloud Notes Application

An iOS based Mobile Notes App

Zachary Kanzhe Liu

Index

[ZaCloud Cloud Notes Application 1](#_Toc447922690)

[Section 1: Introduction 2](#_Toc447922691)

[Section 1: Requirement, Tools and Business Model 3](#_Toc447922692)

[Section 1: Timeline 5](#_Toc447922693)

[Section 2: Research 6](#_Toc447922694)

[Section 3: Challenge 6](#_Toc447922695)

[Section 4: Learning Experience 8](#_Toc447922696)

[Section 5: Future Extensions 9](#_Toc447922697)

[Design Document 10](#_Toc447922698)

[Introduction to Server End 10](#_Toc447922699)

[Server End: Operating System and Database 11](#_Toc447922700)

[Server End: PHP Function Modules 11](#_Toc447922701)

[Introduction to Client Side 15](#_Toc447922702)

[Client Side: Basic Classes 15](#_Toc447922703)

[Client Side: Local plist Copy Management 16](#_Toc447922704)

[Client Side: JSON Connections 18](#_Toc447922705)

[Client Side: Inner Functions 19](#_Toc447922706)

[Client Side: UIViews 21](#_Toc447922707)

[Reference: 25](#_Toc447922708)

# Section 1: Introduction

ZaCloud cloud based notes app is an iOS based app which can synchronize note record across different devices with same login account. This project is conducted by professor Michael Lajoire, and implemented by me (Zachary Kanzhe Liu). This document aims to achieve two sets of objectives. The first set is, to demonstrate the business process and structure of the software, introduce the technologies that are used to achieve the goal, and the development method. Second, present the instruction manual.

This project provided a good opportunity to study and practice the fundamental PHP and Swift programming. It also involves a fair amount of content from Operating Systems, which lead to deep understanding of computer science.

# Section 1: Requirement, Tools and Business Model

The goal of ZaCloud is synchronize notes on different devices owned by the same user. In order to implement the system from a mobile standpoint, there were several issues which had to be taken into account when designing the application:

1. The note records should be stored in both locally and in an online “cloud” service.
   1. A local copy accelerates the data loading and accessing while the online copy preserves the latest version of records.
2. Local client must be able to connect with server through internet.
3. The system requires a rudimentary user management scheme.
4. If internet connection lost, the local client should be able to store the updates and new notes temporarily, and synchronize them with the server after.
5. Deletion will be disabled while internet connection lost.
6. During the edit mode for a particular note, other devices should be blocked out of this note, and reject all of the synchronization from them.
7. Deletion and update require the lock. That means if a device is running offline from the very beginning, it won’t be able to update and anything.

In terms of cover those objectives, the following technologies have been applied for running the business:

1. **L**inux - Amazon EC2 cloud system with Linux Ubuntu 14.04 Long Term Support (LTS). This is the system that can be easily implemented and accessed from anywhere with internet connection.
2. **A**pache 2 Server – Used in order to administer the web page and manage the /var/www/html/ folder in Ubuntu for internet accessing of content
3. **M**ySQL database. The widely used database with common SQL query syntax and stable performance.
4. **PHP JSON**. This is a standard transition format that commonly used for mobile apps internet information exchange.
   1. **PHP** is used to convert MySQL query data into JSON so that the application can properly interpret data from the server
5. **Xcode** with **Swift 2.0**. The standard IDE for iOS app development. It has a powerful simulator and a robust library that helps developer build and test the project effectively.

As the result, the architecture of this software has appeared as the figure down below:

Ubuntu@Amazon EC2

Apache 2 Server

Database:Mysql

Front End: PHP JSON pages

JSON Data Stream

Local client built by swift 2.0

# Section 1: Timeline

In order to finish the project on time, the timeline has been set as follow:

3 January - 9 January: Make most of story board

10 January - 16 January: Design Database

17 January - 23 January: Setup the server side, and try to make Json works

24 January - 30 January: Make sure that php side could return the correct value for client’s query

31 January - 6 February: Aglie program for client side, mostly for control panel

7 February - 13 February: Aglie program for client side, mostly for control panel

14 February - 20 February: Aglie program for client side, mostly for control panel

21 February - 27 February: Aglie program for client side, mostly for control panel

28 February - 5 March: Debugging and researching GCD part

6 March - 12 March: Debugging and researching GCD part

13 March - 19 March: Debugging and preparing the documents

20 March - 26 March: Debugging and preparing the documents

# Section 2: Research

The research part of this project is mainly emphasised on language library functions such as how to convert time from a string, how to retrieve data from database, and encode into JSON. Basically, the research is problem driven activity, and defect of that was I made a bunch of wrong choice, especially for the local storage mechanism.

Through the research of this project, I got the basic view of iOS library and PHP database connection library. These are all focus on the application level, not very into the computer science. There was one research on the system level, which makes sense on performance level. The following chapter will explicit it in detail.

# Section 3: Challenge

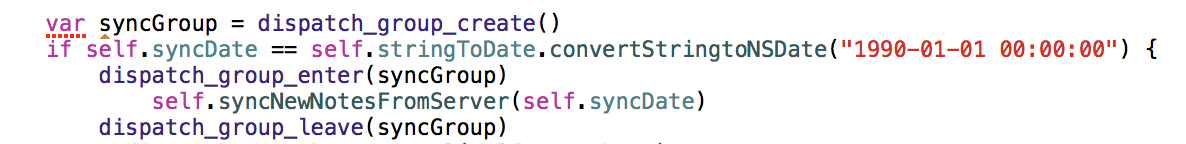
The most challenge part of this program was Grand Central Dispatch on iOS. As we know, Apple products own the “best user experience” title in past ten years, and one of the reason is that those devices have powerful multi threading computation ability. This trait allows an iOS device executes several blocks at the same time. However, it causes a serious programming problem: one thread would get empty result if it depends on the other thread. For instance, if the normal process is download data from network first, then put it into view, the iOS will execute “download” and “load the view from local storage” simultaneously. Since transmission from internet is way slower than local, view will be loaded before downloading complete, and no data will be showed on the screen. As the result, **Grand\_Central\_Dispatch** (**GCD**) plays a very important role in this app, and it is also the most challenging part of this project.

There are two different mechanisms of GCD have been used in this app. The first one is **semaphore**. This one really helps some JSON connection functions that we really wish it runs in the background. At the beginning of the execution, a **dispatch\_semaphore\_signal** rises the signal to hold the resource, so even instructions still be pushed into the queue, the code that right after signal would not be executed unless it sees the semaphore is equal to the semaphore it just been pushed into queue. The signal will be reduced by **dispatch\_semaphore\_wait** when the block is actually done, and system will continue execute the rest of code. This mechanism is also very suitable for the function that needs to make decision according to the feedback from network.



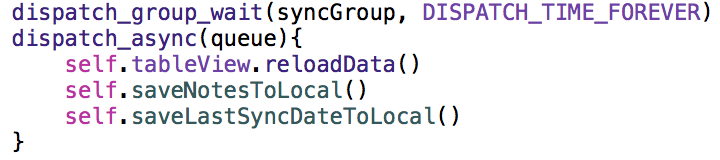
(Semaphore control lines in send function of Locking class. “return value” waits forever until the JSON transmission done, and hits the semaphore\_wait)

The other way to achieve sequential execution is **dispatch\_group**. In synchronization function, multiple JSON connections needs to be done first, and then refresh the whole table view. However, if multiple processes try to get into the **dispatch\_group** and grab the resource at the same time, the thread will be congested and the app will be stuck there forever. As the result, a **dispatch\_group** has to be implemented as asynchronize mode. Here, the program manipulates the **NSURLSession** asynchronized execution trait, we simple put **dispatch\_group\_enter** and **dispatch\_group\_leave** surround each block to indicate the group situation (in the real situation, **dispatch\_group\_leave** should be placed inside of a block at the ending point). Since everything is under monitor, the CPU would not be given a big surprise and hook in somewhere forever.



(dispatch\_group\_create and the dispatching process. This is a demo, not the actual code)

After all of the sync activities done, notes dictionary and a new time stamp need to be wrote to the local plist. Here, the **dispatch\_group\_wait** waits until all of the processes in the group have been done, and grab a predefined queue to execute the write to local task.



(dispatch\_group\_wait and the actual dispatch\_async. In the actual code, the **main queue** has been used for executes rest of the code)

# Section 4: Learning Experience

Through is project, I’ve learnt how to setup the AWS with Ubuntu, basic usage of PHP with Apache2, and Xcode with Swift2. During the development, agile programming and learning have been practiced. I’ve got experience of iOS coding, and got better understanding of threads in Unix system.

There were two major issues I didn’t achieve in this project. First, the core data implementation. According to the Apple document, core data has way better performance than plist that I’m currently using. I didn’t do the research pretty well and choose to follow a tutorial to use plist. The second defect of this project is no research function. Due to the limited time budget, there was not search function has been implemented. The tutorial video is online, and it could be implemented pretty easily.

# Section 5: Future Extensions

This project has been implemented by agile programming, and it was my first iOS project, so there are some obvious issues can be easily found, and the app could be improved in the future.

1. The local plist of notes can be substituted by **core data**. Core data is a database like storage. Compare with plist, it has better read/write performance and advantage object mapping ability.
2. Optimize instruction dispatch. In the future version, **dispatch\_async** could be used for execute parallel tasks. For instance, each of the JSON connection methods could have a private buffer for downloading data, and those methods can be put into multi thread to maximum the usage of transmission bandwidth, and improve the performance.
3. Reduce database redundancy. The current version does not delete anything from database, so it would cause redundancy for a long run. In order to solve the problem, another table that records deletion information could be used, and the note record could be actually deleted from database.
4. Improve synchronization. Compare with Apple iCloud, this system is very rudimentary. It has multiple way to improve: make it like an instant message app (which iCloud does it), or increase sync frequency in the critical time. Likewise, refresh the notes every time that user hits a cell or button.
5. Better internet error protection. In the current version, app has some unhandled internet connection error and causes program crash. In the next version, connection timeout should be added in and prevent exception happens.

# Design Document

In order to present the design and development in a very specific way, the design document will be divided into **Requirement and Business Model, Introduction to Server End, Server End: Operating System and Database, Server End: PHP Function Modules, Introduction to Client Side, Client Side: Basic Classes, Client Side: Local plist Copy Management, Client Side: JSON connections, Client Side: inner functions, Client Side: UIViews, Client Side: Instruction Dispatch,** and **Future Extensions**.

# Introduction to Server End

The storage on the server end of the ZaCloud app is powered by a MySQL database and several PHP pages which sit on the Amazon EC2 cloud. The core functionality of the server involves interacting with the database. Through PHP data is queried and allows the application to, execute selection, deletion, and update. Once this takes place, the PHP modules get the query result, pack the data into JSON format, and pass it through internet. The details of these data transactions will be covered in this Server End section.

# Server End: Operating System and Database

The free trial Amazon EC2 with Ubuntu was been selected for server platform. MySQL database has been installed by updated apt-get, and the PHP pages run on the Apache Lamp. The following two commands were used for this purpose:

../Desktop/Screen%20Shot%202016-03-30%20at%2011.43.54%20AM.png../Desktop/Screen%20Shot%202016-03-30%20at%2011.43.35%20AM.png

The database has two tables: users and notes. The users table stores the **username** and **password**. This project is aimed on design of cloud system, so security issue would not be considering, and none of the passwords have been hashed. The file login.php queries and deals with login activity. The process will be specified in next chapter.

The notes table stores all of the notes with necessary information with it. This table contains attributes such as **title**, **note**, **lastModified**, **createDate**, **author**, **isDeleted**, **lockingTime**, and **lockingDevice**. **CreateDate** and **author** are combinational primary key. In order to maintain the “last version” in the database, deletion operation actually never deletes anything from the database. It simply turns the **isDeleted** to 1 to tell the clients to delete this record in the local copy. It also keeps a possibility that if user update the deleted record and s/he really want to keep it.

The database query syntax has been encapsulated in the PHP pages. The functions of those PHP pages will be explained in the next chapter.

# Server End: PHP Function Modules

The PHP pages run on Apach2 Server. Apache retrieves the url query such like [**http://54.191.76.174/locking.php?author=zach&createDate=2016-3-15+09:07:37&lockingDevice=000334513&lockingTime=2016-1-1+00:00:00**](http://54.191.76.174/locking.php?author=zach&createDate=2016-3-15+09:07:37&lockingDevice=fuck&lockingTime=2016-1-1+00:00:00) to operate certain PHP page, and get the variables from clients. In the previous example above, **54.191.76.174** is the public IP address of EC2 instance, **locking.php** is the PHP file name, and everything after question mark is the variable. Once server received the url query, it assigns the values to local variables through syntax such as **$author = $\_REQUEST['author'];**. Now, server can use those variables to do the further operations.

An operation that server side needs to do is query the database. In this project, **mysqli** library has been used for this purpose.

**$connection = mysqli\_connect("localhost","root","root","projectII") or die("Error " . mysqli\_error($connection));**

Those two lines above are the typical syntax to create an SQLconnection. Once the SQL query such as **$sql** has been created, the syntax:

**$result = mysqli\_query($connection, $sql) or die("Error in Selecting " . mysqli\_error($connection));**

would be used for getting the query result by manipulating **$connection** and **$sql**. If the database responses the query and feedback some result, server can pack the result as JSON format, and echo it to the page that will return to the client.

In terms of order the data in a certain format, server will create an empty array first, then fetch each row as independent variable by using syntax:

**while($row =mysqli\_fetch\_assoc($result))**

at the end, echo the result to return page by using **echo JSON\_encode($emparray);**. As the result, the page will show the JSON data like: **[{"status":"false"}].**

SQL queries that interact with database can just follow the normal sql format by manipulating the variable that is contained in the url. The following paragraphs will explain those queries by functions.

**Login.php:** This PHP page retrieves **username** and **password** from client, then get the actual password from database such as

**$sql = sprintf("select \* from users where username = '%s'", $username);**

It then checks how many rows have been affected and compares the password string is match or not. Users are currently hard coded into the database, so it guarantees no duplicated rows. So, if affected row is equal to 0 or password string does not match, a “**false**” status will be return, otherwise return “**true**”.

**Insert.php:** This PHP page retrieves variables of a new note, and insert those variables into database as a new record. It currently assumes that every insertion would work, and returns nothing back to the user.

**Server.php:** This PHP page retrieves **author**, **lastSyncDate**, and **updateDevice** as variables to get existed note that update after certain date. In order to get existed note, the query sets that **createDate** should before the time stamp, and **lastModified** should after the time stamp. In order to reduce data transition, **updateDevice** also been restricted to not equal to current deivce.

**Deletion.php:** This PHP page retrieves the **author**, **createDate**, **deleteDate**, and **updateDevice** as variables. This operation set **isDeleted** to 1, and update the **updateDevice** and **lastModified.** This means the record is still in the database, so if user still has a way to get it back.

**Update.php:** This PHP page gets all of the variables of a note, and update it to the existed record. It assumes that every update would be executed properly, and it does not return anything back to client. In this version, if a user updates a note that has already been deleted from server side, the server will still accept the update, and inform the other clients.

**newNotes.php:** This PHP pagegets **author**, **lastSyncDate**, and **updateDevice** as variables. It will compare the **lastSyncDate** with “1990-01-01 00:00:00” to see if they are matching. If **lastSynceDate** contains “1990-01-01 00:00:00”, that means this device is a new device for this app, and it will download all of the notes from server. Else, it will query the database to get new notes after **lastSyncDate**. This operation also restricts the server not to query the record that uploaded by current device.

**syncDeletionTable.php:** This PHP page gets **author**, **lastSyncDate**, and **updateDevice** as variables. It takes **lastSyncDate** as time stamp, and get the deletion records from database.

**Locking.php:** This PHP page retrieves **author**, **createDate**, **lockingTime**, **lockingDevice**, and **currentTime** as variables. **lockingTime** that provided by client is two minutes ahead **currentTime**. The server will compare the **currentTime** with **lockingTime** that stored in database, if **currentTime** is beyond the **lockingTime** in database, server will update the **lockingTime** and **lockingDevice** of that particular record (searched by **createDate** and **author**), and return “**true**” to the client. Otherwise, it would not do anything to database, and return “**false**”.

Data will be transmitted back by JSON echo. The rest of the process will be deal in the local client. The following chapters will explain the data process in the local client, and the techniques it would use.

# Introduction to Client Side

The client side merges data from server and data created by users. In order to improve the user experience and minimum the transmit data, local copy and log system will be very important of this app.

The client can be divided into five parts: **Basic Classes**, **Local plist Copy Management**, **JSON Connection**, **Inner Functions**, and **UIViews**. The default execution mode of Apple devices is parallel mode, so the built-in **Grand Central Dispatch** would be necessary to use for organize the instructions. The following chapters will explain each module in specific way.

# Client Side: Basic Classes

Basic classes, sometimes known as the “Model” part of the Model, View Controller System, are the fundamental unit that organize data in the memory. Those classes combined with Swift built-in arrays contain the notes in the dictionary and different kinds logs. all of the basic classes contain data serialize function that convert objects into file stream, and vice versa.

Classes that need to write to the local plist copy need to extend “**NSObject**” and “**NSCoding**”. These two classes provide functions that explicit variable’s “forkey” and its data type. This is very crucial for converting objects to plist items, and mapping plist items back to objects.

**Note** class: This class contains attributes such as **author**, **note**, **title**, **createDate**, and **lastModified**. Client mainly deals with this class object array. It is like a note dictionary, and every new note and update will be modified in this dictionary first. Cell content and text content is loaded from here too.

**User** class: This class only contains a **username**. The reason for use this class is that the index of notes uses UITableView, which is embedded in a UINavegationView, and they cannot run without each other. However, the login view cannot initialize the UITableView. It can only touch the UINavegationView. In another words, the login view cannot pass the user id though segue. In terms of solve this problem, User class has been created for write the current login user to a local plist, and read it back to memory when UITableView has been taken online.

**SyncDate** class: This class contains **username** and **syncDate** variables. It is a time stamp indicator, and it needs to be updated every time when the local client syncs to server.

**DeletionRecord** class: This class contains **username**, **createDate**, and **deleteDate**. It temporarily stores the deletion record from server. When the deletion synchronization operation is done, no deletion record should be remained in memory and plist copy.

**UpdateRecord** class: This class contains **username**, **createDate** and **updateDate**. It records update record that hasn’t been sync to the server. When the update synchronization operation is done, no update record should be remained in memory and plist copy.

**NewNoteRecord** class: This class contains **username**, and **createDate**. It records new note that hasn’t been sync to the server. When the new note synchronization operation is done, no new note record should be remained in memory and plist copy.

Those classes are frequently write to the local plist copy. The next chapter will explain the process of plist management.

# Client Side: Local plist Copy Management

In order to get plist copy of objects, **NSObject** and **NSCoding** classes have to be implemented, and all of the variable types have to serializeble typs such as **NSDate**, **NSString** etc. When system encodes variables into plist items, method **encodeObject** in **NSCoder** class can explicit the forkey of that variable such as:

coder.encodeObject(self.username, forKey: "username")

The coder is an instance of **NSCoder** class. It hands in the result to **NSKeyedArchiver** to save as plist.

In the decoding process, method **decodeObjectForKey** in **NSCoder** decodes plist items, casts into the certain type of data, and assigns them as variables. It can quickly build an instance by using creator that defined in class. Here is the sample of decoder method in NewNoteRecord class:

required convenience init?(coder decoder: NSCoder){

guard let username = decoder.decodeObjectForKey("username") as? NSString,

let createDate = decoder.decodeObjectForKey("createDate") as? NSDate

else {

print("decoding failure from sync date")

return nil }

self.init(

name:username,

createDate:createDate

)

}

The key word “**guard**” has been putted here to handle exceptions.

Before app reads or writes a file, the first thing it needs to do is get the file path. Here, we use author plus fixed file name as the name of a particular file, also **NSSearchPathForDirectoriesInDomains** has been use for find the file directory. The sample code as follow:

let noteName = author + noteListName

let path = NSSearchPathForDirectoriesInDomains(NSSearchPathDirectory.DocumentDirectory, NSSearchPathDomainMask.UserDomainMask, true)

let basePath = path[0]

let noteListPath = (basePath as NSString).stringByAppendingPathComponent(noteName)

The path and data array will be handed over to **NSKeyedArchiver.archiveRootObject** for saving to plist, or **NSFileManager** to decode the data.

Every basic class has a plist read, and a plist write methods. They have been intergrated into **NotepadIndexTableViewController** class.

# Client Side: JSON Connections

Local client has a communication method for each PHP page. The process of each method are quite similar, and here is the explanation of those JSON Connection methods.

As this report mentioned in PHP Function Modules, the server retrieves a certain format url query. So the first step of connection process is get the right url which decorated with correct variables. Here, the url string picks the fix part of the address, then sets the variables from the object, and expends the url string. Since space is not allowed to appear in a url query, the url string also needs to convert in UTF8 format. Here is a sample of it:

let myUrl = NSURL(string: "http://54.191.76.174/insert.php")

let request = NSMutableURLRequest(URL:myUrl!)

request.HTTPMethod = "POST"

let postString = "title=\(note.title)&note=\(note.note)&author=\(note.author)&createDate=\((note.createDate.description as NSString).substringToIndex(19))&lastModified=\((note.lastModified.description as NSString).substringToIndex(19))&updateDevice=\(UIDevice.currentDevice().identifierForVendor!.UUIDString)&isDeleted=0"

request.HTTPBody=postString.dataUsingEncoding(NSUTF8StringEncoding)

After url has completed, **NSURLSession** will take over the rest of the job. The method **dataTakeWithURL** sends the url to server, and get the result data. If the result data is not nil, then all of data we get can be casted into String, and assigned to matching variable by forkey.

According to the Apple document, **NSURLSession** default in asynchronized way to execute, and it always runs on background thread. Since transmitting through internet takes way much more time than local data exchanging, an oblivious consequence would be like the the empty view has been loaded completed and be posted on the screen, while the data is still on the way. There are multiple ways to solve the problem. In this project, two mechanisms, **dispatch\_group** and **semaphore** have been applied for fix the issue. These two techniques will be mentioned in Instruction Dispatch chapter.

Local plist copy management and JSON connections are the core of the local client. All of the operations on business tire are combined by them with some other inner functions’ assistance.

# Client Side: Inner Functions

There are two kinds of inner function in this program: assistance function, and complex function that consisted by fundamental functions.

**DateTimeFromString**: This class converts a certain format string to NSDate type by using **NSDate** and **NSDateFormat**. It also prevents the exception caused by nil date like “0000-00-00”.

**LockingTimeGenerator**: Method in this class is a helping method of SendLocking class. It generates a NSDate time which 2 minutes ahead of current time. If server accepts a certain lock requirement, this time object will be the expire time.

**SendLocking**: This class contains two methods, send and unlock. Those two methods have the same mechanism. It shoots the current date time along with locking date time. There is a slightly difference between these two, send method sends the lockingTime which two minutes ahead of current time, and unlock just sends current time as lockingTime, so it would expire in next second.

**isConnectedToNetwork**: This method tests the connection to the server, and return a Boolean value. It manipulates a Swift built-in function **NSHTTPURLResponse** to send a request, and judge the network situation by status code.

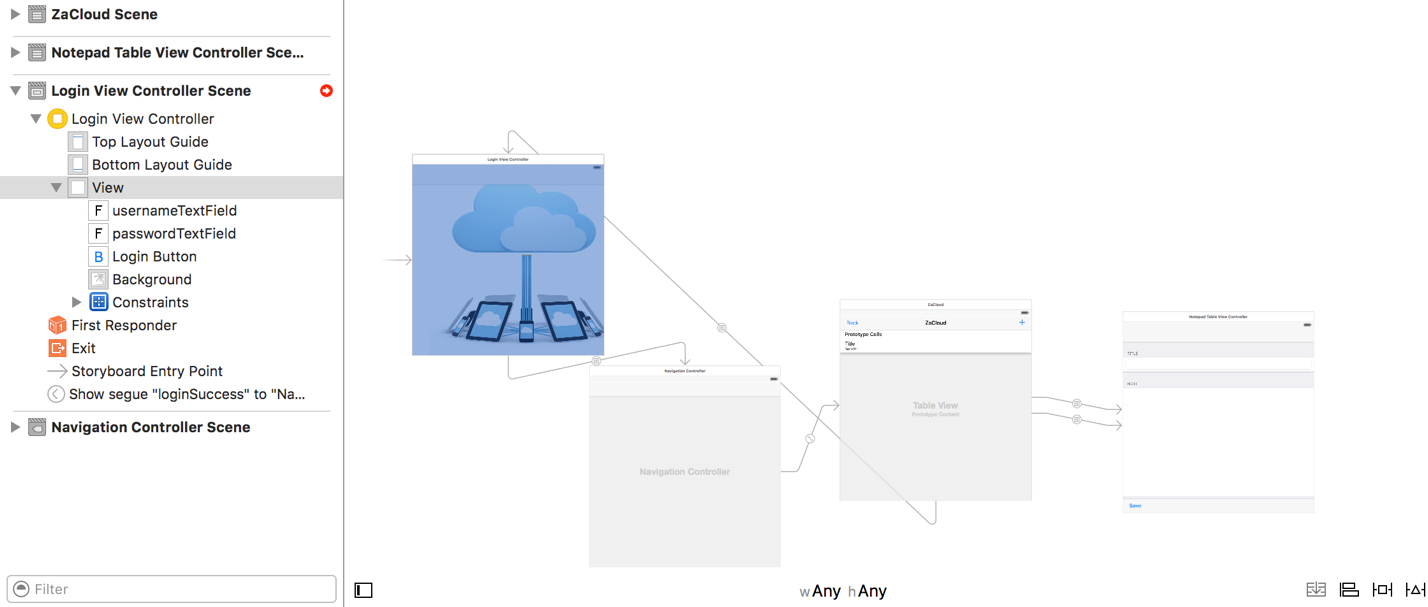
**Synchronization**: This is a complex function that syncs with server. It first syncs all of the un-updated notes to the server, and downloads all of the notes that have been changed after last sync date. Then, it syncs all of the deletion records from server, and makes the modification to local. The last thing it needs to do is sync new notes with server. The new notes just simply added to the end of the array. This method involves local plist management, JSON connections, and most of helping method.

**loadData**: This function is used for initialize **NotepadIndexView** when it has been first loaded. It first uses **loadDefaultUserFromLocal** to get the author, then **loadNotesFromLocal** to initialize the notes dictionary with local copy. After that, it tries to synchronize with the server. In the end, it refreshes the whole view to set the data into cell.

**Login**: This method sits in the **LoginViewController** to operate the actual login activity. It retrieves username and password from text fields, and sends them to the server to get the verification result. It uses JSON connection to login.php, and it also writes the username to a plist to pass it to the **NotepadIndexTableViewController**.

The whole business tire is for showing the correct information to the user. The next chapter will explicit the mechanism of presentation tire in this app, how to setup the information in the correct spot.

# Client Side: UIViews

The development method on Xcode is also called “storyboard driven”, due to its powerful storyboard function. Normally, all of the views have been created on the storyboard, and been setup the appearance. Also, the transit between each view, “segue”, is also been set and named here. Developers really get the benefit from this “what you see is what you get” style. 

(storyboard in Xcode)

The functions and traits has been connected to each view after they have been setup. Classes that control the view are extended from built-in class **UIViewController** or **UITableViewController** that especially for cell table. Apple has already built up a collection of methods to setup and control the views, and the following content will explain these by each view.

**LoginViewController**: The LoginView is the start view. In storyboard, three functional components have been setup there: **usernameTextField**, **passwordTextField**, and **LoginButton**. **LoginButton** has a segue to the **NotepadIndexTableViewController**, **usernameTextField** and **passwordTextField** have layout in the controller for accessing purpose. In the controller, a method “login” to run the activity, and other methods and objects are for support it. Due to the background may block text fields and button, **viewDidLoad** method which provided by **UIViewController** has been override. It runs a function called **sendSubviewToBack** to push background to the position that it should be. After user typed in the correct information and clicked the **LoginButton**, the segue way to the **NotepadIndexTableViewController** may be triggered. In order to determine the login segue should be trigger or not, function **shouldPerformSegueWithIdentifier** has been override, and been put **login** function inside of it. This function returns a Boolean value to determine segue should be performed or not. In another way, it is a mechanism that decides login is success or not.



(login view)

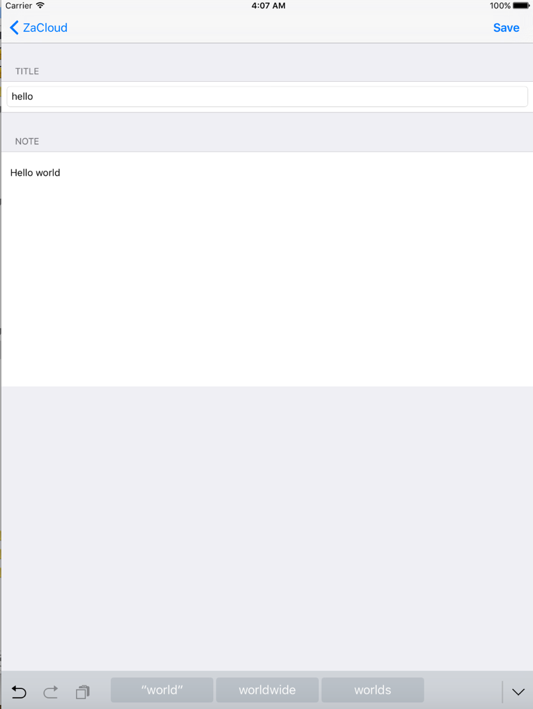
**NotepadIndexTableViewController**: **NotepadIndexView** is the main view that browse the notes. There are **cells**, **backButton**, and an **addButton** in the **index table view** which sits in a **NavigationControllView.** When the loadData function finishes initialization of notes dictionary, an override function **tableView** which return **UITableViewCell** will put notes’ title and last modified date to each cell. This function is controlled by **tableView.reloadData** function, this function resets each cell every time it been called. Once user clicks the **addButton** or any of non-empty cell, the function **prepareForSegue** will choose to setup the **titleTextField** and **noteTextField** or leave it empty by **segue.identifier**. the delete function hides in slide the cell to left. This operation passes **commitEditingStyle** to tableView, and triggers the delete function. If user clicks the delete function, that particular note will be deleted, and controller will **reloadData** after that.



(Notepad Index View)

User can also drag down the whole table view to synchronize with the server immediately. Here, controller make a **UIRefreshControl** variable to define its behaviour. If user does this gesture, the function **handleRefresh** will be triggered by the **UIRefreshControl** variable, and do the jobs that be wrote in the function.

**NotepadTableViewController**: **NotepadView** is where user edits the note content. It has **titleTextField**, **noteTextField**, **saveButton**, and **backButton**. The **backButton** is just simply return back to index, **saveBotton** saves notes to local and server, **titleTextField** and **noteTextField** are the editing area. If user does not get the lock, **saveButton** will be unavailable. If user gets it, the controller will refresh the lock every 110 seconds (The locking time is 120 minutes) to keep the lock.



(Note View)

The **saveButton** function sits in the **NotepadIndexTableViewController**. A function called **unwindSegue** deals the saving mechanism. It gets an **UIStoryboardSegue**, casts a variable as **sourceViewController** of the segue, then retrieves data from source view and save it to local and server.

# Reference:

1. Mr. Daniel Friyia’s Xcode class
2. Apple Swift Tutorial: Simple iOS9 Swift Application with Spotlight Search: <https://www.youtube.com/watch?v=BH_V8a5xKGk>
3. tableView Delete Swipe in Xcode 7: <https://www.youtube.com/watch?v=XtzHEvsJYmE>
4. unwind segue: <https://www.youtube.com/watch?v=KzVShJnffIs&list=PLe92GDCP3_05iyDc89jY5W_HotZ4A81Sm>
5. Control flow: <https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/ControlFlow.html>
6. PHP retrieve data from JSON: <http://www.kodingmadesimple.com/2015/01/convert-mysql-to-json-using-php.html>
7. How to make REST API calls & parse JSON with Swift 2.0: <https://www.youtube.com/watch?v=uQ_MyVDiSbo>
8. php insertion: <http://www.kodingmadesimple.com/2014/12/how-to-insert-json-data-into-mysql-php.html>
9. insertion on swift side: <http://swiftdeveloperblog.com/send-http-post-request-example-using-swift-and-php/>
10. NSCoding/NSKeyedArchiver for local storage: <http://nshipster.com/nscoding/>
11. Php: mysql\_affected\_rows(): <http://php.net/manual/en/function.mysql-affected-rows.php>
12. swift subString: <http://stackoverflow.com/questions/24904738/how-to-use-substringtoindex-in-swift>
13. swift network status: <http://stackoverflow.com/questions/24516748/check-network-status-in-swift>
14. swift pull to refreshing: <https://www.andrewcbancroft.com/2015/03/17/basics-of-pull-to-refresh-for-swift-developers/>
15. swift async dispatch: <http://stackoverflow.com/questions/19822700/difference-between-dispatch-async-and-dispatch-sync-on-serial-queue>
16. semaphore in swift: <http://stackoverflow.com/questions/28522853/semaphore-does-not-wait-before-executing-block-of-code-after-it>
17. swift timer: <https://www.hackingwithswift.com/example-code/system/how-to-make-an-action-repeat-using-nstimer>
18. dismiss view controller: <http://stackoverflow.com/questions/24668818/how-to-dismiss-view-controller-in-swift>
19. Dispatch groups: <https://www.raywenderlich.com/79150/grand-central-dispatch-tutorial-swift-part-2>