Bachelor Degree Project

Automatic synchronization and data merging between iOS and server databases

- Solution for easy setup of synchronized offline capable ***crud*** *functionality between application and server*

Author: Mikael Melander
Supervisor: Johan Hagelbäck
Semester: ST 2018

Abstract

Keywords: iOS, crud, Linux, offline synchronization

Preface

Contents

1 Introduction

This degree project will study and develop a solution for an iOS project with server integration to handle offline data synchronization and data merge control.

The study aims to examine the best rules or algorithms for data merge solutions as well as data structures that can be handled by both iOS and an online server database solution.

1.1 Background

The world of mobile applications has exploded in the last couple of years. Companies increasingly adopt the functionality of mobile devices to streamline the daily workflow, what used to be done by pen and paper or only by sitting down at a computer is now possible with mobile applications.

Because a mobile phone can be taken anywhere, your work should be able to follow, but most applications demand an external database to store data across users, this, in turn, relies upon a mobile data connection or WIFI by the device itself.

This can become a problem if you find yourself in places where your cellular reception is limited. If this happens then you won't be able to work. The solution to this is to be able to add, see and use the data that already exists within the application regardless of your cellular reception.

1.2 Related work

Research around similar solutions have been conducted and there are some big and famous companies that have created some type solutions to the problem. These are some solutions that work really well and integrates the solution that this project aims to solve. They have some drawbacks and exactly how they solve it isn't disclosed.

Firebase is Googles database service that allows you to create a mobile/web application connected to the database within a couple of steps. They have a complete framework for iOS, android, windows phone and web solutions like javascript and more. They also have a set of tools to verify data, create security, analyze usage, send push notifications and many more. Firebase as a Google solution locks you down to their own way of thinking, you need to use it on their premises or not at all. You can only store data in their database and build your solution around them.[1]

Microsoft Azure also supports the ability for offline client sync, but as well as Firebase it relies on you using their cloud services[2]

Within this area, there are a lot of different solutions that solve the problem, but they solve the problem within their particular code language and platform.

Some examples of this are solutions that work for frameworks such as Xamarin that is a cross-platform .NET solution, react-native that is a cross-platform JavaScript solution and Ionic that is a JavaScript cross-platform solution that works with html5 instead of native as react-native does[3][4].

These solutions work with specific server platforms, and some of them only locally duplicating the database that is online, not handling the merging situations that the server needs to be able to handle. Most solutions that I found aim towards integrating an offline synchronization solution for an already existing platform, as a layer to support the platform. What this project aims to do is to integrate the whole server and offline integration natively from the start and then build the app upon that.

1.3 Problem formulation

The goal is to find a suitable solution that is a reusable starting point for developing a mobile application within iOS that connects to a server and database backend. A project that already has the functionality of connecting all of the server, database and iOS frameworks.

The finished project should be able to query, edit and upload data locally/offline and automatically handle the upload/query to the online server database.

We need to develop and define a database structure that correctly keeps track of data versions. This will be needed because the system needs to be able to handle different data versions being received and sent by different users/devices that have manipulated data while offline.

To be able to handle these data merges we also have to develop a set of rules or algorithms to handle the merges correctly. The rules need to support multi-tenant usage and should not corrupt any data.

1.4 Motivation

Today it already exists different solutions and platforms for offline data synchronizations.

Like Googles Firebase, it is a fully online platform that solves creating database structure, merge rules and querying data, but using it locks you down to their backend and systems and you have little to no control over your data.

Many companies today that want a solution for mobile applications associated with their workflow would not allow that the data is stored anywhere other than in their own control. This to not give away any company information or users information to a third party.

So the creation of a ready to go server/iOS solution that runs on an environment that is able to be deployed to any server and have offline functionality already implemented would be a huge timesaver.

This as well as having a solution that can be further developed over time the more you work on similar projects.

1.5 Objectives

O1	Research and determine server platform, data structure and language
O2	Implement connection and upload/query functionality to the server database
O3	Implement local storage iOS
O4	Implement methods for querying data given specific arguments that handle both local/server database
O5	Implement functionality to keep track of data versions
O6	Determine and perform experiments for data merge rules

The big goal of the project is to find out if you can create an offline synchronization solution between server and iOS device that is usable instead of the big corporation solutions, a solution where you are in control of all the aspects of the data. As well as find the merge rules for the data that is the most commonly used to be able to build a solution that is actually usable as a real replacement.

To be able to achieve this we need to find a server framework that is free to use and can handle at least one of the same database language that Xcode can handle. To get the offline functionality how to keep track of data versions.

We then need to implement the functionality to connect and upload/query data to the server database. When that is done we can implement the same database structure locally on the iOS side.

When we have both a local and server database working we can implement functionality for iOS to automatically upload/query data depending on the cellular/WI-FI connection.

For it all to work together the functionality to keep track of data versions and merge rules needs to be implemented to the server side.

1.6 Scope/Limitation

Within the scope of this project, the solution should allow for an as minimalistic setup that's possible, that gives you a reusable solution within Xcode that would allow you to query data and return results, either from locally saved data or from the online database depending on your internet connection.

You should be able to save new data and edit previously added data that when connected to an internet connection should be synced with the online database.

The iOS project should be written with Obj-C and because Xcode and iOS changes over time the solution should be set to work and support at least the latest 3 versions of iOS (9-11) which is the version the majority of iOS users use.

The solution that is created and the dependencies used for it has to be free.

This project will not take into consideration the security aspects meaning that it will not have a solution for HTTPS or that it will have any users or data access lists.

The project will not support real-time database.

1.7 Target group

This project can be of interest to companies, organization or persons wanting to be in control of their own data and host their own solutions that integrate with iOS in a cost-efficient way.

The solution should also be considered as an open starting point to keep building upon, but that already has the important implementations for server integration and offline data support.

1.8 Outline

The next chapter will present the **methods** that are used to execute the different objectives that are presented above.

The **implementation** chapter will be a more detailed explanation of how the project will be implemented and how the solution itself works, how the merge rules will be executed and how the automatic syncing is handled.

The **result** chapter will cover what came of the project, what the resulting structure of the solution became.

The **analysis** will cover an overall analysis of the concluded results.

The **discussion** will deeper discuss the analysis and results.

Chapter seven will include **conclusions** that are based on the results as well as present future work and recommendations.

2 Method

The method used to conduct this project will be verification and validation.

The project is not created in any collaboration with a company, meaning there is not a given outline from an external source to create these requirements. So to get the requirements for this method, the defined problems for the project will be converted into functional and non-functional requirements.

This project does not build upon already existing code or will not use any existing code that will have to be collected (This does not include the frameworks and platforms that will have to be used). This means that the functionality of the project will be based upon written code for the functions that need to be implemented, so the requirements will make sure that the implementation and functionality are correct.

By using the verification and validation method we can see if the project supports the functionality that is required by verifying and validating the requirements with different manual tests that are connected to the required part of the implementation.

2.3 Reliability and Validity

To be able to use the verification and validation method correctly and be sure that the results are reliable, the requirements we create needs to be measurable and objective. This means that we need to make sure that the requirements can't be subjective if they are subjective different people can interpret the requirements in different ways. If this would happen the reliability of the results could be compromised.

The requirements created from the problem definitions will be broken down in small pieces that will be easier to understand and phrased in a way that should be easy to confirm or deny if it is fulfilled or not. For example "Is the data saved locally if no internet connection is available?", it should be fairly easy to answer yes or no. Conducting it in this way will help to ensure the reliability that the verification and validity are correct.

The verification and validation method is most often used when you want to confirm the results of a working project to a customer, but because the project is conducted by only one person the validity of the results might be questioned. Therefore there is an even bigger reason the requirements needs to be as simple and direct as possible.

This project aims to create a solution that will be continued being developed after this project. There would be no reason for the results of the verification and validity to not be correctly conducted.

2.4 Ethical Considerations

The project goal is to create an easily deployable server, iOS and offline data synchronization solution. The solution should be open sourced and has a potential to be worked on more to create extra functionality and widen the scope.
By conducting this project there should not be any reason for any ethical issues to come to light.

3 Implementation

This chapter will explain how the solution was implementation and conducted. It aims to explain a bit of the technicality.

3.1 Goal

The plan for the complete solution is an iOS client application working with the built framework.

The framework needs to be able to have full CRUD functionality with the local database, to return the result to the user for faster loads and offline capability. Then depending on internet access send the CRUD request to the server.

The server then validates the data version and executes the merge rules, saves the correct data to the database and sends a response back to the framework.

The framework now updates the local database if needed and sends another response back to the client. This can be seen in Figure 3.1.1.

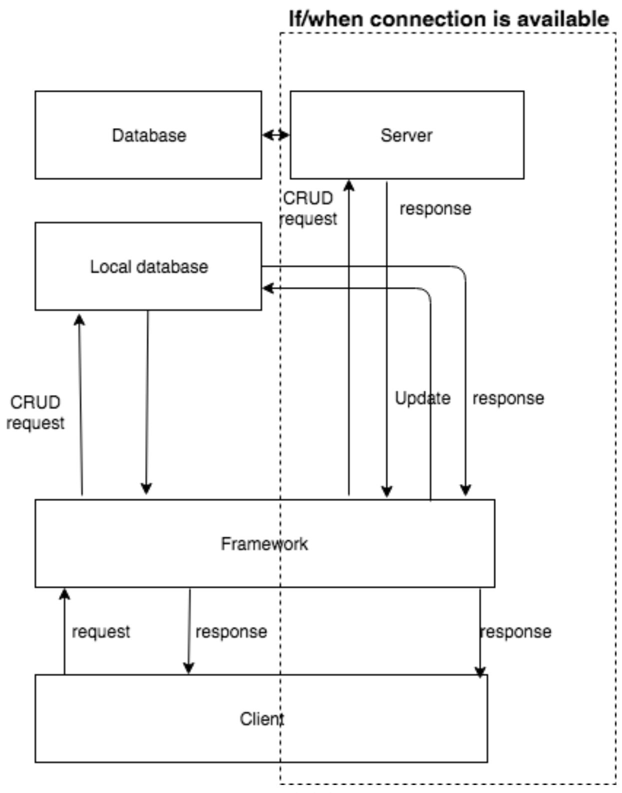

Figure 3.1.1 Design

3.2 Database

Before the server framework could be determined the database language needed to be set.

The choice was set between MySQL and MongoDB and decided based on the pros and cons selected in table 3.2.1 below**[7]**.

MySql	**Pros:**
	Mature
	Support Join
	Privilege and password
	Native iOS support
	Cons:
	Stability concerns
	None community driven
MongoDB	**Pros:**
	Integrated storage engines
	Dynamic schemas
	Cons:
	No native iOS support
	Young solution

Table 3.2.1 Relational database

The decision for this solution landed on MySql. MySql has been around for a long time and a lot of people are familiar with it, but the biggest reason was that the native support already exists within iOS.

3.3 Server implementation

To determine what server framework to use, there were some restraints. The framework had to be free, and support the database type that was decided earlier in **3.2**.

Below in table 3.3.1 are some of the considerations and the important pros and cons of the server framework **[6]**:

Node js	**Pros:**
	Fast
	Full Stack
	Lightweight
	Big open source library

	Cons:
	Unstable API
	Less fitting for CPU intensive tasks
Ruby on rails	**Pros:**
	Flexible
	High quality (because of set standards)
	Evolved framework with a lot of tools
	Consistent
	Cons:
	Slow
	Large stack frame
	Depends on Apache/Nginx or something similar.

Table 3.3.1 Server framework

The decision landed on Node Js.

Node js is a free solution that is widely used across the world and uses JavaScript which includes Npm. Npm is used as a collaborative community that gives you free access to re-usable code. This in term is optimal for this project since the goal is to make it open source and people can keep building upon it.

The complete server solution is built as a REST API with full CRUD functionality, together with the MySql database. The server receives an http call with the CRUD request from the client and compares the data received with the data in the current database if it exists. Then updates the database according to the validation and merge rules. It then sends back a response to the client framework as seen in figure 3.3.2 below.

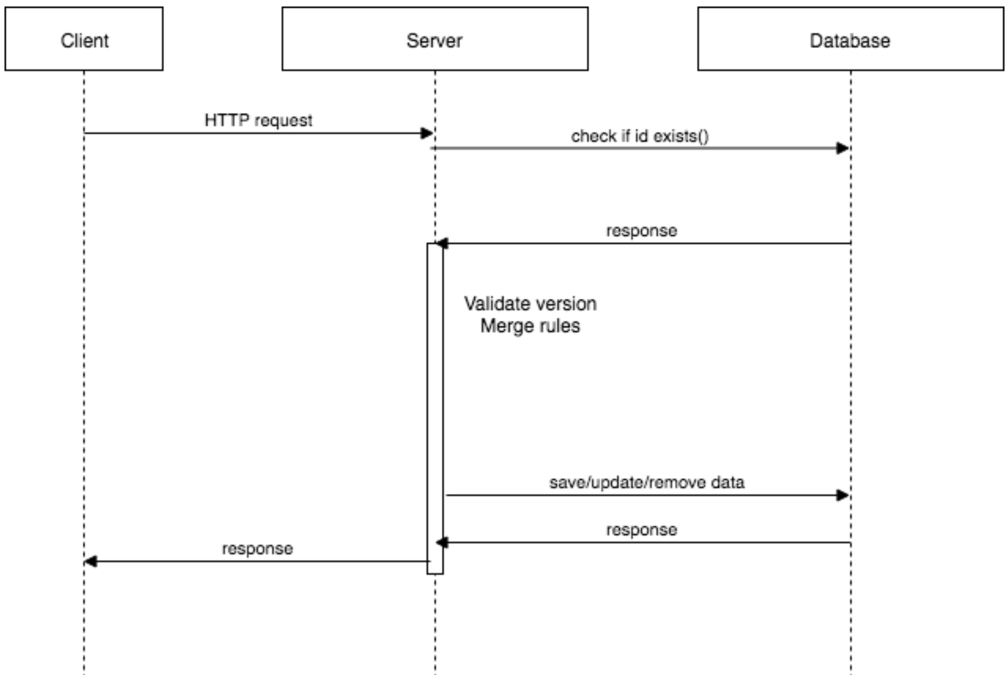

Figure 3.3.2 Server framework

3.4 iOS framework

The goal of the iOS framework is to more or less work as a middle hand that handles all the logic between the server/databases and the user.

It should provide a set of ready to use functions to query, save, update and delete data.

The framework should always query the local database first, to keep the query as fast as possible. When the client is served with the local database data that can be used while offline or to render the UI before the online data has responded, then the framework should check the online database and return the response data to the client.

The same goes for when the framework saves data, the data is saved locally first and presented to the user. Then in the background sent to be saved to the server. If there is no connection available, the background job will be paused until a connection returns, then sent to the server.
This process is shown in figure 3.4.1 below.

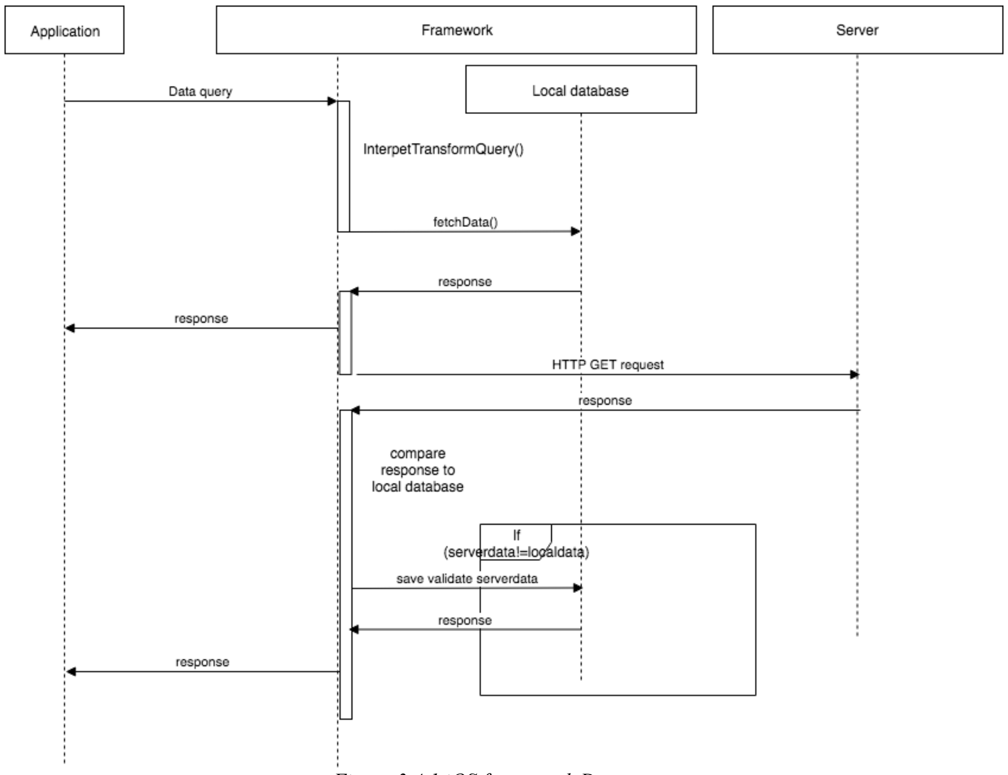

Figure 3.4.1 iOS framework Data query

3.5 Validation and merge rules

The validation and merge rules are the rules the server applies to decide what data is the correct one to keep in the online database.

Because several users or the same user with different devices might manipulate, update and delete the same data while some are offline and some are online the data might exist in several versions at the same time on different devices. To decide what data is the one to keep in the database there needs to be some rules.

The rules considered and implemented follows in the table 3.5.1 below.

Name	**Description**	**Comment**
Newest wins	The most recent data timestamp always wins.	This might become a problem because the updated time will be set on the device. If the user decides to change the time manually on the phone.
Server wins	Synching offline data to the server will be disregarded	For applications wanting to use the framework without the offline update version.
Client wins	Ignores the conflict and changes are overwriting any value in the online database.	This approach might lead to the loss of important data changes
User decides	The user is told about the conflict and gets to decide	This approach was **removed** from consideration because the framework should work with as little effort as possible from users.

Table 3.5.1 Validation and merge rules table

The validation and merge rules are implemented on the server side before the save occurs, but the choice of what rules to set is decided within the framework of the application.

Some columns in the data that should always be present and implemented by the framework itself to be able to follow the rules above, seen in Table 3.5.2 below.

createdAt **[timestamp]**	Locally creates a timestamp on the iOS device on the creation of the item
updatedAt **[timestamp]**	When the object is created the timestamp is the same as createdAt but this is the one that updates on each change
synced **[BOOL]**	Only saved in the local database to keep track of what data is saved online and what isn't

4 Results

The result chapter is based on the tests of the final product.

4.1 Requirements

The results are based on tests that are conducted in an iOS simulator on a MacBook Pro while the server is running on the same MacBook in a Docker environment.
The following requirements are converted from the projects Problem description.
Below in table 4.1.1 are the requirements presented.

Requirement	Description	Test
1	Connection to the server by only IP	Manual
2	Saves data **offline**	Manual
3	Query data **offline**	Manual
4	Edit data **offline**	Manual
5	Remove data **offline**	Manual
6	Save data **online**	Manual
7	Query data **online**	Manual
8	Edit data **online**	Manual
9	Remove data **online**	Manual
10	A working set of merge rules	Manual
11	Synchronize local database change to an online database	Manual

Table 4.1.1 Requirements

All the requirements passed and examples will be presented below.

4.2 Frameworks and platforms

The frameworks that were selected to run as the server and database can be seen in the table 4.2.1 below.

Platform	Framework chosen
Database	MySQL
Server	Node.js

Table 4.2.1 Frameworks results

4.3 Merge rules

The results below will be shown for the different merge rules implemented.

The framework will convert the input data to a MySQL insert. The table 4.3.1 below is based on the example user input for a save action (The test was conducted with more than one example, but with results accordingly). The framework will automatically add two columns, as seen in the response, the columns represent timestamps for createdAt and updatedAt as well as the ID column that will always be represented as a 1 in these examples.

The table 4.3.2 and table 4.3.3 will show the different responses for an update example according to the merge rules, the server response is always the **winning** data, that then **always** will be saved as local as well.

Input	Local response	Server response
(Växjö-Kalmar, 0,0)	(1, Växjö-Kalmar, 0, 0, 2018-04-29 11:21:01, 2018-04-29 11:21:01)	(1, Växjö-Kalmar, 0, 0, 2018-04-29 11:21:02, 2018-04-29 11:21:02)

Table 4.3.1 Save data outputs ***According to requirements 2 and 6*** *(No merge rules on save data)*

Examples for newer than updatedAt data **All data sent 11:30:00**		
Current data online and offline for all examples	(1, Växjö-Kalmar, 0, 0, 2018-04-29 11:21:02, 2018-04-29 11:21:02)	
Input data for all examples	(1, Växjö-Kalmar, 1,0, 2018-04-29 11:21:02, 2018-04-29 11:30:00)	
Rule	**Local response**	**Server response**
Newest wins	(1, Växjö-Kalmar, 1, 0, 2018-04-29 11:21:02, 2018-04-29 11:30:00)	(1, Växjö-Kalmar, 1, 0, 2018-04-29 11:21:02, 2018-04-29 11:30:01)
Server wins	(1, Växjö-Kalmar, 1, 0, 2018-04-29 11:21:02, 2018-04-29 11:30:00)	(1, Växjö-Kalmar, 0, 0, 2018-04-29 11:21:02, 2018-04-29 11:21:02)
Client Wins	(1, Växjö-Kalmar, 1, 0, 2018-04-29 11:21:02, 2018-04-29 11:30:00)	(1,Växjö-Kalmar, 1, 0, 2018-04-29 11:21:02, 2018-04-29 11:30:01)

Table 4.3.2 Edit data outputs. ***According to requirements 4, 8 and 9*** *(DRAFT)*

Examples for older than updatedAt data All data updated 11:30:00		
Current data online and offline for all examples	(1,Växjö-Kalmar, 0, 0, 2018-04-29 11:21:02, 2018-04-29 11:45:05)	
Input data for all examples	(1,Växjö-Kalmar, 1,0, 2018-04-29 11:21:02, 2018-04-29 11:30:00)	
Rule	**Local response**	**Server response**
Newest wins	(1,Växjö-Kalmar, 1, 0, 2018-04-29 11:21:02, 2018-04-29 11:30:00)	(1,Växjö-Kalmar, 0, 0, 2018-04-29 11:21:02, 2018-04-29 11:45:05)
Server wins	(1,Växjö-Kalmar, 1, 0, 2018-04-29 11:21:02, 2018-04-29 11:30:00)	(1,Växjö-Kalmar, 0, 0, 2018-04-29 11:21:02, 2018-04-29 11:45:05)
Client Wins	(1,Växjö-Kalmar, 1, 0, 2018-04-29 11:21:02, 2018-04-29 11:30:00)	(1,Växjö-Kalmar, 1, 0, 2018-04-29 11:21:02, 2018-04-29 11:30:01)

Table 4.3.3 Edit data outputs for offline data update earlier than server stored data. ***According to requirements 4, 8 and 9*** *(DRAFT)*

Table 4.3.4 below will show and explain some alternative scenarios and what the result of the situation.

Rule	Scenario	Local response	Server response
Newest wins/ Server wins	Attempting to remove a local object that has been updated more recently on the server.	Removes object	Returns the updated object
Client Wins	Attempting to remove a local object that has been updated more recently on the server.	Removes object	Removes object
All	Attempting to update an object that has already been deleted from the server	Returns error, object still removed	Returns error, object still removed

Table 4.3.4 Alternative scenarios. ***According to requirements 5 and 9***

5 Analysis

The goal of the project was to create a server, online database and iOS framework to handle automatic synchronization between data and support offline capabilities within the application. When looking at the results in chapter 4 of the report they support the knowledge that this is feasible within the project that was created.

The result shows that the solution supports the main use cases that the merging rules are designed for.

The local data will always first conform to the users' input data, it will be able to edit, save and delete data so that the framework is always usable even if the device doesn't have any reception. But because the createdAt/updatedAt local data sent to the server will conform to the devices own time settings this can become somewhat of a problem. If you would to manually change the time on your device the data sent to the server would be handled with that timestamp, this could become a problem.

Then depending on the selected merge rules the data change accordingly when an internet connection and data synchronization is initialized. This shows that the project would be able to replace the bigger cloud solutions if you need an offline synchronized capability for iOS but still want control of all your data yourself. And even if you don't want the offline capabilities it will still be an open sourced project that already implements a functioning database solution for iOS.

Because it's running on a node.js server and is open source this project could expand in the open source community and create several additional functions and purposes which was one of the final goals of the project.

6 Discussion

The biggest reason for this project, to begin with, was the possibility to show that having an open sourced backend to handle the situations that are the biggest reasons for companies and individuals to turn to the big companies for their ready to go solutions isn't as needed as most people think.

The reasons to use their backends is the functionality that they have ready to go. Having a solution that can be open sources might give the community a solution that can over time implement more and more of the functionality given by these companies. There are several reasons to do this, for example being in control of the data stored by self-hosting the solution. And in this time with the new GDPR laws, this looks more and more attractive. And working as a consultant for other companies they might demand a self-hosting solution.

6.1 Validity of results

The test comes from the requirements of the project and is seen in chapter 4. They were conducted by manually inputting the requests in the code on the iOS side and running it. It was tested several times and with different data and the results was as expected.

The only part that can compromise this data is that it has only been tested on a node.js Docker, running on the same computer as the iOS simulator. Although the result shouldn't vary because of this, but the solution hasn't been tested on a self-hosted server.

7 Conclusion

In the thesis, the goal was to try to find out if it was possible to create a possible self-hosted replacement for the current solutions like Firebase or Azure. The reason is to keep your data in-house and not give it to big third party companies. The scope of the project didn't for obvious reasons include all the possibilities that these big expanded solutions already have, but more to build an open source project that handles the part of offline synchronization and merge rule handling.

This functionality has been implemented and tested according to the set requirements and within the scope, the project has been proven possible.

And because the project is running on an node Js backend the possibilities to keep implementing open source libraries makes it fairly easy to keep adding functionalities.

I would like to think that the project came to the appropriate result according to the set goal. It's proven that it's possible to build an offline synchronized solution that is reusable and is a good start to keep building on in future work.

7.1 Future work

The time limitations of the project made the scope narrow against what is possible.

The big picture of this project would include an Android framework to integrate with the backend, continues functionality to the backend as for example Push notifications, file upload/download, analytics, remote config, a/b testing and the list can go on forever. The implementation possibilities of the backend and is almost endless.

In the more current picture, the security aspect of the backend would be implemented, user sessions/login and database protection for injection attacks for example. I would also like to see a more script-based setup to improve usability.

References

[1] **Firebase: Offline capabilities on iOS**
https://firebase.google.com/docs/database/ios/offline-capabilities
Accessed: *2018-02-29*

[2] **Azure: Enable offline syncing with iOS mobile apps**
https://docs.microsoft.com/en-us/azure/app-service-mobile/app-service-mobile-ios-get-started-offline-data
Accessed: *2018-02-29*

[3] **OpenSoure: Does your app work without an internet connection**
http://opensourceforu.com/2017/01/mobile-app-without-internet/
Accessed: *2018-03-01*

[4] **Github: SQlite-Sync.com version 3**
https://github.com/sqlite-sync/SQLite-sync.com
Accessed: *2018-03-18*

[5] **Frontmag: Offline data synchronization in Ionic**
https://frontmag.no/artikler/utvikling/offline-data-synchronization-ionic
Accessed: *2018-03-18*

[6] **Medium: NodeJS vs Ruby on Rails…**
https://medium.com/@TechMagic/nodejs-vs-ruby-on-rails-comparison-2017-which-is-the-best-for-web-development-9aae7a3f08bf
Accessed: *2018-04-10*

[7] **MongoDB vs MySql**
https://hackernoon.com/mongodb-vs-mysql-comparison-which-database-is-better-e714b699c38b
Accessed: *2018-04-09*