Hotei: Backend Services Component Report

Nick Robertson, Department of Electrical and Electronic Engineering, Imperial College London

***Abstract*— To fulfil Hotei's aim of helping manage a student's overall well-being through the provision of tailored activity recommendations, a recommendation system is required. This recommendation system requires an interface for each user's application to securely send their activity data to a backend server. This server will perform collaborative filtering [1] and return personalised suggestions to each user. The focus of this report is on the communication between the application, security standards, and architecture of the server.**

I. INTRODUCTION

The high pressures associated with pursing further education is having consequences on the mental health of students. One study found that 27% of all students suffer from a mental health problem, and 63% of students report suffering from stress levels that interfere with their day to day lives [2]. Hotei aims to reduce students overall stress levels by offering tailored recommendations as and when their connected wearable device detects a state of elevated stress, or they request one.

A crucial component of Hotei is hence the recommendation of suitable activities based on a users' previous responses to activities. This report does not consider the intricacies of generating recommendations but rather considers the process of transporting and storing the information from individual users and and returning a set of personalised recommendations to the application.

To generate recommendations all Hotei users' activity data is required. A database presents itself as a clear solution for storing users' personal data. Before designing this database these questions must be considered:

- How can the mobile application communicate with the server?
- What measures will be taken to ensure users' data is stored securely?
- What type of database architecture is best suited for the context?

The report seeks to answer these questions by examining existing technologies and research in the field. This investigation will provide the basis on which design decisions will be made for Hotei's backend services.

II. BACKGROUND AND RELATED WORK

The architectural style for the backend services is an important part of designing the Application Programming Interface (API) for communicating with the server. It determines the communication style, performance characteristics, and compatibility.

A. RESTful Services

REpresentational State Transfer (REST) is an approach to providing communication between separate systems through the internet. RESTful Services are not a new idea, Roy Fielding laid out his original idea for RESTful services in his PhD dissertation in 2000 [3]. Since publishing this paper, RESTful services have become a staple choice for mobile and web developers due to its valuable intrinsic properties.

RESTful Services are based on 6 fundamental principles:

1) *Client-Server*: There is a distinct layer between what is a server and what it a client. A client component is one with a set of requests, a server component offers a set of services and listens for requests from the client. Hotei fits this model well as the application fulfils the role of the client and the backend is the server.
2) *Stateless*: No state information is held on the server between requests, hence each request must contain all of the information necessary to understand the request. This fits with the current requirements of Hotei as the server does not hold any information about the client's state between RESTful calls.
3) *Cacheable*: If data is cacheable the client should cache the information, reducing the required network communication. In Hotei, the recommendations for the user are a prime candidate for caching. The application should cache this information as their recommendations are unlikely to change drastically in short time periods (once issues of cold start have dissipated).
4) *Uniform Interface*: Simplifies the overall system architecture as well as interactions between components. Uniform interfaces require identification of resources, manipulation of resources through representations, self-describing messages, and hypermedia as the engine of application state.
5) *Layered System*: Layers are used to encapsulate legacy services and to protect new services from legacy clients. Intermediaries can be used to improve the system's potential scalability primarily through means of load balancing. In the context of the project, traffic should not be an issue however it is useful future proofing of the application.
6) *Code on Demand*: Client functionality can be extended via the use of downloading code such as applets and scripts. Currently this is not entirely relevant however if the decision is made to move from iOS to a native web application (eg. Cordova based [4]) this could be helpful for enhancing *live* content.

Given the widespread nature of REST APIs there are many companies that have gone to the trouble of providing pre-built solutions [5]. This raises the question whether it is a worthwhile endeavour writing Hotei's mobile backend given the availability of Software as a Service (SaaS). The decision has been made by the group that the API will be written in-house as it is preferable to being constrained by an external vendors' APIs; Additionally writing in-house gives more control over hardware choices allowing local resources to be used for testing rather than paying for cloud services.

B. GraphQL

A new alternative query style that is being described at the future beyond RESTful services is Facebook's GraphQL [6]. GraphQL enables the client to ask for exactly what they need rather than getting superfluous information or having to make multiple REST calls. GraphQL is based on these

design principles: hierachal, product-centric, client-specified queries, backwards compatible, structured, arbitrary code execution, application-layer protocol, strongly typed, and introspective. [7]

A paper written by Kit Gustavsson comparing REST and GraphQL aptly states that "using REST results in a much more flexible and adaptable solution while using GraphQL is more restrictive for the developer. GraphQL also needs an implementation that sets some restrictions on which languages that are possible to use" [8]. In his paper he goes on show that GraphQL helps to reduce required traffic between the client and the server due to most descriptive calls.

As fascinating as it would be to develop using this young language, high levels of traffic are not of particular concern due to the limited and constrained nature of communication between the server and the application. The flexibility and and well supported nature of RESTful services make it the preferred choice for Hotei.

C. Data Protection

The protection of data is of growing concern in the context of mobile healthcare. In this report the protection of data will only be considered in the context of transit between the app and the server and being stored in the server. The on-device security is beyond the scope of this report. The European Union's move to increase the laws surrounding the protection of personal data [9] makes it of paramount importance that Hotei provides a sufficient level of security, such that if the application were to be released it would be compliant with European data protection standards.

The data in transmit is at risk of man in the middle attacks. A means to mitigate this risk is to encrypt the data. HyperText Transfer Protocol (HTTPS) is a protocol for secure communication on the internet. It is an extension of HTTP which makes use of Transport Layer Security (TLS), the successor to Secure Socket Layer (SSL), to encrypt the packet. The connection uses certificates and asymmetric keys in order to exchange a symmetric key with which to secure the subsequent information [10].

An additional layer of encryption that could be considered is SecJSON [11]. This additional layer of security encrypts the actual JSON data rather than the entire packet. To implement this additional layer of security would be excessive in addition to HTTPS and would be needless overhead, as well as the risk of adopting a method that is not an industry standard. For these reasons it is not going to be included in Hotei's communication protocol.

Ensuring the security of the data on the server is also highly critical in protecting personal information. Database security encompasses three main properties: confidentiality, integrity, and availability [12]. Confidentiality is assured by making sure that only authorised users can view data, in the context of Hotei, a user can only read and change their own data through the strict API. Integrity of the data is maintaining that the data is correct, this is achieved through the database management system. Availability is ensuring the data is available when users request it. Some do not consider this basis enough and demand that even with a bitwise copy, a hacker cannot interpret the data. It is not realistic to use Transparent Data Encryption (TDE) for the entire database due to the overheads associated with handling the encryption. It is more realistic to consider Column-Level Encryption which encrypts a particular column in the database table. The columns with personal data that could be used to identify a user could be encrypted as they are not frequently accessed and do not pose a credible risk of causing bottlenecks.

Differential Privacy is a method to help protect against an insider attack. It stops a trusted party from using the dataset [13], [14]. It introduces noise to the data to anonymity data whilst also ensuring that it does not affect the statistical significance of the information on display, hence enabling a machine learning algorithm to detect the same relationships. This would be a useful stretch goal of Hotei however it is lower down on the list of objectives as it is not considered a high risk component at this stage of development.

D. Storing Big-Data

Hotei needs to store the data associated with all users on a central database. To begin with this should be easily achieved however if the number of users and inputs recorded grows steps need to be have been taken to ensure that the solution scales well. *Traditional* databases systems for storage are based on the relational model. These are SQL databases, named after the language they were queried in [15]. With the prevalence of big data the popularity of NoSQL databases has grown due to their horizontal scaling properties.

Relational databases were propose by E.F. Codd in 1970 [16]. The model is simplistic organising data into a table of columns and rows with unique keys identifying each row. The benefits of this model is that data can be organised across tables using linked rows (foreign keys). Relational databases are strong candidates for scaling vertically. In the context of the proof of concept application the simplicity of the implementation, the static nature of the data being handled, and the expected small dataset for the project lends itself to using a relational database.

NoSQL is moniker adopted by non relational databases. It is not table based and it is instead document based. It is less structured data with a more dynamic schema available for handling this. As it is not strictly required to follow the ACID principals of relational databases it make is easier to horizontal scaling that is beneficial for large datasets. From an intensive investigation between the performance characteristics of key-value stores of NoSQL and SQL one paper found that in general NoSQL had better performance characteristics however it varied from each type [17]. The ability of this type of database to perform better and scale is tempting however it is slightly more complicated to set up.

Although it might make sense in the future to change to NoSQL, at this point in Hotei's life cycle, SQL seems to be more than capable of handling the data. Additionally the data is not modified particularly regularly by users with anticipation reads and writes with periods in unit of hours or days. For the testing it is likely that it will all be run on a single server so it is not logical to invest the additional time required to implement a NoSQL solution at this time.

III. SYSTEM DESIGN

A. Design Overview

The database that will persist the data produced by Hotei will be server based, this choice stems from these design decisions;

- The activity recommendation system is dependent on the information of other users to provide tailored recommendations, having a centralised location for the data makes analysis more convenient.

- The amount of processing power available is far greater and more scalable than that on a mobile device
- It will enable users to access their account from multiple devices

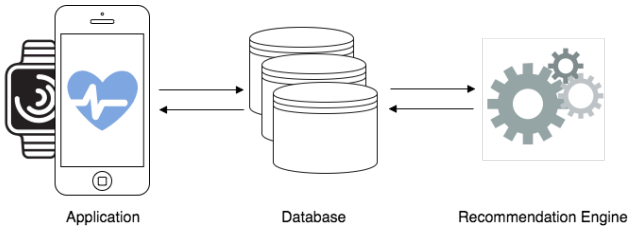

Fig. 1. Simplified Overview of Hotei's Architecture

There will be RESTful API for a database backend. The server will be hosted using Microsoft Azure, as they strive for excellence in data protection and privacy, which is a necessity in health applications. To protect the RESTful API from external attacks, no API calls can be made without authentication, every user will have an access token before being able to read or write data. Then all information in transit will be encrypted using HTTPS. Figure 2 shows the flow of a user entering the system and receiving data from the API.

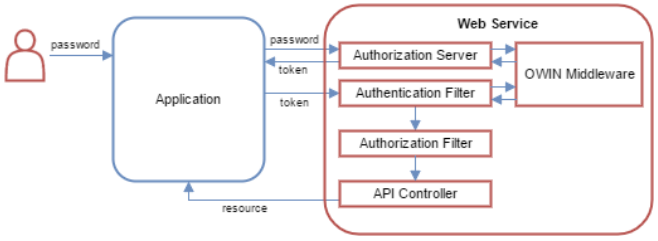

Fig. 2. User API encryption mechanism

The language of choice for this development is C# and its SQL interface. The decision was made due to the recommendation engine developer's familiarity with the language. A purist would argue that it is suspect combining an Apple front end with a Microsoft backend however the beauty of the HTTPS RESTful API is it's prevalence in the industry leading to cross platform support.

B. Database Organisation

As mentioned in the prior section, it made sense in the context of the project to pursue a relational database model. The tables of interest will be Users, Activities, Reviews, and Recommendations. Users will store information relating to the each individual user. Activities will hold of the potential activities and particular information on them. Reviews will contain data reviews of users' activities. The exact nature of the recommendations table has not yet been decided upon given the result of the recommendation engine is similar users rather than activities. It is hence assumed in this situation that rather than storing the nearest neighbours, the activity suggestions of the nearest neighbours will be considered. The ER Model in Figure 2 and the tables help to clarify the organisation of the content (N.B. Not all columns are finalised and are subject to change when more inter-component testing occurs)

TABLE I
USERS

UserID	First Name	Last Name	DOB	Gender
000001	Jeremy	Johnson	12/01/1964	M
000002	Lisa	Smith	01/03/1996	F
000003	Joe	Bloggs	30/08/1994	M

TABLE II
ACTIVITIES

ActivityID	Activity Name
000001	Kayaking
000002	Rowing
000003	Wind-Surfing

TABLE III
REVIEWS

TimeStamp	UserID	ActivityID	Score (/10)
114512932019	000001	000001	7
114543932019	000001	000004	8
114512931932	000001	000002	6
114512931839	000002	000002	2
114591282938	000003	000002	3
114597657778	000003	000003	7

TABLE IV
RECOMMENDATIONS

UserID	ActivityID
000002	000004
000003	000001

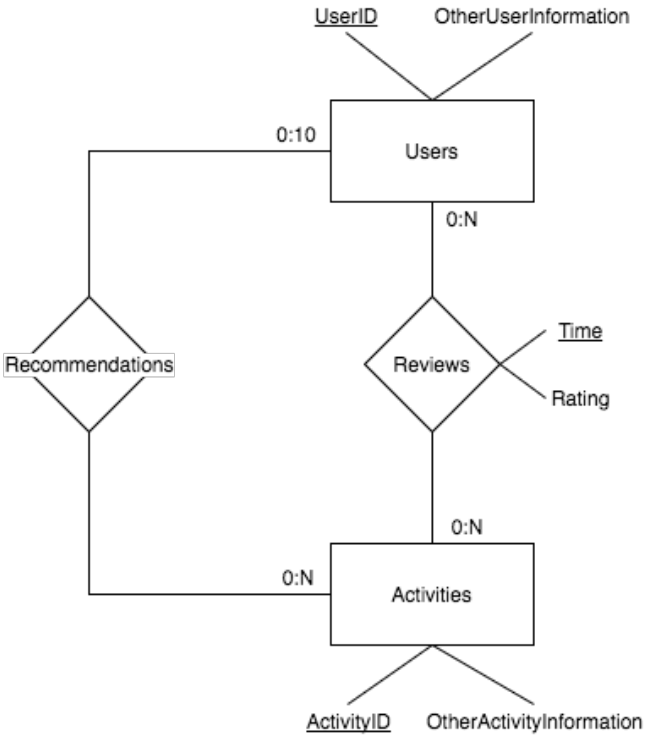

Fig. 3. Database ER Model

C. API Design

The number of RESTful calls required is small. The current design aims for minimalism and considers only the most basic calls required. These calls all assume that the user has been authenticated and thus authorised to access the resources.

- *Register User*: Creates a new user in the users table, this requires generation of a new UserID.
- *Delete User*: Removes a user from the user table. At this point in time there has not been a decision as to whether all associated records will be removed as it would be beneficial to keep them for the sake of the recommendation engine.
- *Modify User*: Updates field inside the users database, shouldn't affect anything else as the UserID is the only member field that can't be changed and it is the only member field that affects other tables.
- *Get Recommendations*: Gets the current recommendations on the server system for the user. The activity recommendations are returned in JSON format. It is likely that this will only be called by the user a few times a week as it can be cached and the results are unlikely to change drastically in short periods of time.
- *Get Activity Information*: Get review for activities such that they can be stored locally on the device. The primary purpose of this call is for the scenario in which the use has multiple devices or has changed to a new device. It is not expected to be called frequently.
- *Put Activity Information*: Adds a review to the review table for a user. It will be able to send multiple review simultaneously to reduce overhead of communications as well as better suiting the batch communication strategy.
- *Modify Activity Information*: Changes a review, expected to be called in the scenario that user input error.
- *Delete Activity Information*: Remove a review that was added by the user accidentally. This is not a method aimed at removing all a user's activities, such a call would be better suited to being place on a queue and executed at a time of low activity.

IV. IMPLEMENTATION

A. Back-end Services and Testing

To check that the proposed tables and keys were correct a simplistic MySQL implementation was created using dummy data to check that the underlying design was correct. This was a quick prototype that could be tested manually using a set of queries to check if the response were as expected. With the relational database model working, a C# implementation using the SQL Database with RESTful interface was built. This was a very simple step thanks to Microsoft's comprehensive templates. This was not hooked up to the recommendation engine and was only the code for the database and the API. The API calls specified in the design are working and can send and receive the appropriate JSON files. The testing was completed locally making use of Postman, a Google chrome plugin, and cURL.

B. Future Work

Further testing of the API needs to take place to check that it works with the Apple environment. The initial steps have been taken to set up a development environment to test this however due to time constraints the testing has not been completed. It is anticipated that there should not be any issues in communicating from the application due to the universal nature of HTTP calls.

No form of authentication has been implemented at this point in time as it requires more work with the front-end developers to ensure it is agreed upon how a user will register an account. It would be preferable to use a technology such as OAuth 2.0 to allow users to login with a Facebook or Google account as this would make handling the authentication on the server side simplistic as it is well supported in C# environment.

Currently the server has only been tested on a local machine and has not been placed on the Azure cloud. This will be required to test Hotei as a whole, it will be completed in the future.

C. Unforeseen Challenges

The boundaries between each individual components is somewhat blurred and it's difficult to separate the decision making as the consequences of a decision in one component can have knock on effects in another. In the future the team needs to improve communication to ensure that any consequential decisions are discussed. An example of this is the recommendation engine which was deemed a separate component of Hotei to separate the work, however in actuality it sits on the same box hence interoperability between the two has required close work with the developer.

V. CONCLUSION

In conclusion, the backend services of Hotei are a critical piece in the overall architecture, connecting the application data to the recommendation engine. A wide range of sources had to be consulted in order to determine the design approach to take however a relational database system with a REST API was decided upon. The fundamental pieces of the implementation are working however at the time of writing the security of the API is not complete due to a lack of authentication. The next stage of the project is bringing the individual components together and beginning overall system testing.

REFERENCES

[1] J Ben Schafer, Dan Frankowski, Jon Herlocker, and Shilad Sen. Collaborative filtering recommender systems. In *The adaptive web*, pages 291–324. Springer, 2007.

[2] Matthew Aronin, ScottSmith. One in four students suffer from mental health problems, 2016.

[3] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[4] Apache. Apache cordova, 2015.

[5] G. Liang, L. Mei, S. Li, and L. Chen. Exploiting unique characteristics of mobile backend services to recommend right ones. In *2015 IEEE International Conference on Mobile Services*, pages 464–467, June 2015.

[6] Facebook Open Source. Graphql, 2015.

[7] Nick Schrock. Graphql introduction, 2015.

[8] Kit Gustavsson and Erik Stenlund. Efficient data communication between a webclient and a cloud environment. 2016.

[9] European Commission. Protection of personal data, 2016.

[10] Alfred C Weaver. Secure sockets layer. *Computer*, 39(4):88–90, 2006.

[11] Tiago Santos and Carlos Serrão. Secure javascript object notation (secjson) enabling granular confidentiality and integrity of json documents. In *Internet Technology and Secured Transactions (ICITST), 2016 11th International Conference for*, pages 329–334. IEEE, 2016.

[12] Luc Bouganim and Yanli Guo. Database encryption. In *Encyclopedia of Cryptography and Security*, pages 307–312. Springer, 2011.

[13] Jianqiang Li, YANG Ji-Jiang, Yu Zhao, Bo Liu, Mengchu Zhou, Jing Bi, and Qing Wang. Enforcing differential privacy for shared collaborative filtering. *IEEE Access*, 2016.

[14] Cynthia Dwork. Differential privacy: A survey of results. In *International Conference on Theory and Applications of Models of Computation*, pages 1–19. Springer, 2008.

[15] Kevin Kline, Daniel Kline, and Brand Hunt. *SQL in a Nutshell: A Desktop Quick Reference Guide*. " O'Reilly Media, Inc.", 2008.

[16] Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[17] Yishan Li and Sathiamoorthy Manoharan. A performance comparison of sql and nosql databases. In *Communications, Computers and Signal Processing (PACRIM), 2013 IEEE Pacific Rim Conference on*, pages 15–19. IEEE, 2013.