Nanyang Technological University

School of Computer Engineering

Final Year Project

Final Report
Android application for cloud-based healthcare monitor

Author:
Tuan Phong Nguyen

Supervisor:
Assoc Prof Chiew Tong Lau

May 9, 2016

Abstract

There is an emerging need for proactive healthcare monitor that is enabled by the development of portable devices and Internet of Things. Based on the related work on the network of sensor devices to collect medical data, this project aims to complete the solution by implementing a layer that allows users to store, view and share their data in a reliable and distributed cloud-based system. In this project, a backend system hosted on cloud computing service and an android mobile client was developed. Google App Engine was selected as the platform for server due to its low cost and reliable media to connect to large number of users. On the other hand, the android application was completed as a proof of concept of how the data can be presented to the users in a friendly and convenient way while following the latest guidelines in android development by Google.

Contents

1 Introduction

Along with the increasing popularity of compact and portable platform for measuring health related data, there are many products on the market that are capable of reading and sending data to a computer system. These devices are often built ready for communicating with mobile devices via Bluetooth to exchange data so that those data can be displayed visually. However, a universal platform to collect and display the data to users has not been implemented yet. On the other hand, the measured data is not frequently recorded so that it can be used as reference in diagnosing or study. In addition, many hospitals and clinics has to face the challenge of storing and managing the huge amount of medical data of patients in different forms that are not efficient and reliable. There is such a need to create a platform where the medical data of patients can be stored and viewed easily by patients and doctors. This project aims to implement such a platform with consideration on capability of adapting multiple types of devices, data sharing as well as the scalability. The scope of this project is to implements a cloud-based working application that consists of a backend server for storing data and an Android application that allows users to query their own and other users' data. Overall, the scope of the application can be described with the following features:

1. Users are able to post or submit the medical record (data) to a persistent database (database)

2. Users are able to view their own data as numerical listing and graphical representation

3. Users are able to request to view other users' data and view all authorized data sets simultaneously

4. Users are able to accept a request to view their own data or cancel an authorized request

5. Users are able to receive notifications on their phone about certain events such as emergency or critical data timely.

2 Related work

In the last decade, it is observed that healthcare is shifting from traditional reactive approach to proactive approach so that the issues can be discovered and treated at an early stage. This approach requires a solution to continuously collect sensor data and monitor through a reliable service. With the development of technology and medical system, there are several studies to create a system to monitor patients? condition in real-time effectively and correctly to enable proactive monitoring [1] with the help of embedded computer system [2]. Furthermore, devices with communication capabilities have been developed to measure human body's reading for long-range medical healthcare system and avoid the cost of nursing expense in families [3]. On the other hand, the system which provide a common interface for large number of users must be able to serve great amount of traffics and interactions with the clients. In order to handle large volume of simultaneous interaction to the database, an efficient and scalable approach must be used for database handling instead of traditional server. A highly available and reliable distributed platform is provided by Google Datastore, which offers many possibilities in developing such a system to store the data on the cloud [4]. Superior communication between healthcare professionals is now possible thanks to the advancement of wireless sensors and Internet of Things. Such a development helps the healthcare system to provide a better service by allowing the scientists to access the bigger sample size and the users to utilize the collected sensor data without expert knowledge [5]. To extend the advancement to ubiquitous home healthcare, array of sensor devices to detect Electrocardiogram (ECG) and Photoplethysmography (PPG), blood glucose and ear temperature was developed for Home Integrated Health Monitoring in [6] and in [3]. Data are collected from the devices around the house via ZigBee communication technology, and from those devices, transmitted to home server and then medical institutions. However, viewing and interpretation of the data by the users is still not properly implemented. On the other hand, cloud computing is an emerging trend that offers several advantages to the traditional physical servers and hosting. Google Cloud Computing platform can reduce the cost of large-scale clusters and fault tolerance server to extremely low [7]. The solution for healthcare monitor can be implemented based on Google Cloud Computing stack and use its messaging service, Google Cloud Messaging to send the notifications to the large number of client devices. Although in real world application, the Google

Cloud Messaging are observed to have unpredictable time delay of message arrival, it is capable of delivering messaging to significantly large number of subscribers in a reasonable timeframe. Hence, Google Cloud computing and google Cloud Messaging is a suitable infrastructure for an application that have random accesses by large user base [8].

3 Architecture design

The application contains 3 main components namely the client, the backend server and the database.

3.1 Client side

The client is responsible for displaying the data acquired from the physical measuring devices visually as graph or reading listing. The client application needs to acquire the data records from the backend server through network communication and cache it in local database for presenting to the app user. In order to server as a universal interface to various types of medical record, the interface is capable of displaying different types of graph for different record type and designed to be extended when there is need for a new representation of data. The client connects to the back-end to interact with other users of the application to share and access their data in the database. The client also provides the user with an interface to search for another users of the application and interact with them. Users can subscribe to another user to view their data and receive notifications upon certain events are triggered such as when new data is uploaded or when the reading is beyond safe range. As the database is accessed through a RESTful API via HTTP request and response, the client application can be implemented in any platform e.g. Android, iOS, native desktop program or web application. Previously, there were many medical monitor devices that are capable of connecting to smart phones via Bluetooth and hence required such a platform to posting the data upstream. But with the introduction of various small-size computing module that is network capable (Bluetooth, Ethernet) such as Raspberry Pi 3 or Intel Edison, the link between measuring devices and the server can be completed easily with an affordable cost. Due to time constraint and man power of this project as well as the extensibility to other projects, the client is only implemented for Android phones.

3.2 Server side

The server is a Web server that exposes a RESTful API to the clients and serves the incoming requests. Such a RESTful API that communicates via HTTP request and response is a standard way for network application and provides an abstraction layer between front-end and back-end, allowing a wide array of client devices to communicate with the server using standard message. To perform the business logic, the server interacts with a persistent database to provide the service as application backend. The server side is chosen to be hosted on Google App Engine cloud service so as to elevate scalability and reliability of the application. By using cloud computing from a credible service provider, the application aims to minimize down-time and bottleneck comparing to self-hosting service on a single computer. In addition, hosting application on a cloud platform also enables the application cost to scale better with the number of users and data traffic since the service provider offers multiple pricing tiers for different amount of traffic. Hence, the development stage which does not require large volume of transactions can make use of the free tier and scale up once the application acquires certain user base and transaction volume. Moreover, Datastore database provided by Google also supports caching and reliably distributes data to different locations with minimal latency. Regarding implementation of the server program, since client and server communicate through a RESTful API, the server can also be implemented in any programming languages. In this project, the web server is implemented in Java with Google Endpoints API [?] with Cloud Search API [10]. This approach helps promote code reuse by sharing Java object between client and server codebase. In the future, the server can be swapped out by any web server if there is need for finer control or performance improvement.

3.3 Database

Google Cloud Datastore is chosen to be the database backend of the system to reliably store and deliver data. Google Cloud Datastore is a schema-less NoSQL Datastore providing robust, reliable storage for the web application. In contrast with SQL-like database, Datastore treats each record in the database as a key-value pair without unnecessary indexes. Comparing to the traditional relation databases, the Datastore uses a distributed architecture to automatically managing scaling to very large data sets. Additionally, Datastore is hosted as part of the Google App Engine platform, hence is fully managed with no planned down time by Google. Due to its different way of representing and managing data, the Datastore can be easily scaled, allowing the application to maintain high performance as the traffic is increased.

4 Implementation

4.1 Google Cloud Datastore

4.1.1 DataRecord

The `DataRecord` class is used to contain the information of a user's reading values uploaded from the client application or hardware. The model contains various properties about the data record including the type e.g. Simple, Blood Pressure data, created date and the reading value represented by a String. The representation of a reading depends on the type of interested measurement, hence the Datastore Entity does not enforce any constraint on the String format and relies on the client and server application to correctly serialize and deserialize the value from and to the correct string. `DataRecord` has the child - parent relationship with a user entity that supports efficient querying of all instances that belong to a given user. Lastly, the type attribute is marked to be indexed by Datastore engine so as to the server can query on specific type of data of interest.

Listing 1: DataRecord properties

```
@Entity
public class DataRecord {
    @Id public Long id;
    @Parent private Ref<HealthDroidUser> user;
    private Date date;
    @Index private int type;
    private Date createdAt;
    private String value;
}
```

4.1.2 HealthDroidUser

The `HealthDroidUser` class is used to represent a user of the application separately from their Google account. The `HealthDroidUser` instance is created when the user logs into the system for the first time and will be used as the reference for all their relevant data. The user object will not be created again on the next time the user logs in.

4.1.3 RegistrationRecord

The `RegistrationRecord` class is used to represent a device that logged into the application and ready to receive pushed message via Google Cloud Messaging. The `RegistrationRecord` instance is created when a device logs into the system and removed from Datastore database if it fails to send the message to the device i.e. the device is no longer used or signed in with another user.

4.1.4 SubscriptionRecord

The `SubscriptionRecord` class is used to represent a subscription relationship between a user (subscriber) and another user (target) and create a many-to-many relationship among users. The `SubscriptionRecord` is created upon receiving a request to subscribe from the client application and deleted upon receiving a request to unsubscribe. Since such a request need to be approved by the subscribed party, the `SubscriptionRecord` contains a `isAccepted` value to indicate the status, which is false by default and only set to true once the user accepts that. `SubscriptionRecord` has a child-parent relationship with the target user, and a reference to the subscriber. Thus, the instances can be queried when finding users who subscribed to a given users or users subscribed to by a given user.

Listing 2: SubscriptionRecord

```
@Entity
public class SubscriptionRecord {
    @Id Long id;
    @Index private Boolean isAccepted;
    @Index private transient Ref<HealthDroidUser> subscriber;
    @Parent private transient Ref<HealthDroidUser> target;
}
```

4.2 Server side

The application server is implemented with Google Cloud Endpoints API in Java. The server can be accessed through a RESTful API via HTTP requests and responses. In this project, several models of the application can be interacted with through separate classes called an endpoint, each represent a feature and relevant methods. Each endpoint consists of one or more methods annotated with `@ApiMethod` annotation and is exposed to outside accesses.

4.2.1 Data

`Data API` is contained in `DataEndpoint`. The Data API contains the following methods

Method	Description
addData()	this method is used to add a data entry to datastore. The method is only allowed to be access by authenticated user as required in user parameter
getDataRecord()	this method is used to get the data from datastore. The method takes 2 optional arguments namely userId, to specify the user which data entries belong to and after, to set a lower limit of date in data entries. The ”after” argument can be used to efficiently get only the new data entries in datastore for update operation

Table 1: Data API

Listing 3: Data API

```
@ApiMethod(name = "add")
public DataRecord addData(@Named("value") String value,
   @Named("date") Date date, @Named("type") int type, User user)
@ApiMethod(name = "get");
public List<DataRecord> getDataRecord(@Nullable @Named("userId")
   String userId, @Nullable @Named("after") Date after)
```

4.2.2 User

User API provides methods to interact with the user model of the application.

Listing 4: User API

```
@ApiMethod(name = "add")
public HealthDroidUser addUser(User user)

@ApiMethod(name = "get")
public List<HealthDroidUser> getUser(@Nullable @Named("userId")
    String userId)

@ApiMethod(name = "query")
public List<HealthDroidUser> queryUser(@Named("queryString")
    String queryString)
```

4.2.3 Registration

Listing 5: Registration API

```
@ApiMethod(name = "register")
public void registerDevice(@Named("regId") String regId, User user)

@ApiMethod(name = "unregister")
public void unregisterDevice(@Named("regId") String regId)

@ApiMethod(name = "listDevices")
public CollectionResponse<RegistrationRecord>
    listDevices(@Named("count") int count)
```

Registration API provides the methods to register and unregister a device to the backend system for receiving notifications via Google Cloud Messaging.

Method	Description
addUser()	this method is used to add a user to datastore. The method is only allowed to be access by authenticated user as required in user parameter. Upon user signing in, the method is called and user is created if not already existing in the database. The method is idempotent and being called multiple times does not recreate the user. In addition, the method also create an entry used by Google Search API to support searching with Google's API.
getUser()	this method is used to get the user from datastore. The method takes a userId as String and return information about the data from the database
queryUser()	this method is used to query a list of users from part of the user ID. Since Datastore does not index all attributes by default and hence not allow to straight forward search for user ID string. To support the query for large number of users in the database, the method uses Google Search API.

Table 2: User API

Method	Description
registerDevice()	this method is used to register a device to the backend. The method is only allowed to be accessed by an authenticated user and create a `RegistrationRecord` entity.
unregisterDevice()	this method is used to unregister a device to the backend.
listDevices()	this method is used to query a list of all device registrations that belong to a user.

Table 3: User API

4.2.4 Subscription

Subscription API provides the methods to interact with subscriptions between users in the datastore.

Listing 6: Subscription API

```
@ApiMethod(name = "subscribe")
public SubscriptionRecord subscribe(@Named("target") String
    target, User user)

@ApiMethod(name = "accept")
public SubscriptionRecord
    acceptSubscription(@Named("subscriptionId") Long
    subscriptionId, User user)

@ApiMethod(name = "list")
public List<SubscriptionRecord> listSubscriptions()

@ApiMethod(name = "subscribed")
public Collection<SubscriptionRecord> getSubscribed(User user)

@ApiMethod(name = "subscribers")
public Collection<SubscriptionRecord> subscribers(@Named("userId")
    String userId)

@ApiMethod(name = "pending")
public Collection<SubscriptionRecord> pending(@Named("userId")
    String userId)

@ApiMethod(name = "unsubscribe")
public List<SubscriptionRecord> unsubscribe(@Named("userId")
    String userId, @Nullable @Named("target") String target)
```

Method	Description
listSubscription()	this method is used to list all the subscription records in the database.
getSubscribed()	this method is used to get all the users that is subscribed to by a give user.
subscribers()	this method is used to get all the users that subscribed to a given user.
subscribe()	this method is used to subscribe to a user.
unsubscribe()	this method is used to unsubscribe from a user.

Table 4: Subscription API

4.3 Client side

4.3.1 Overall design

The android application is implemented with the following components:

1. SignInActivity that handles signing in of users
2. MainActivity that serve as the hosting activity for different views of the application with a Navigation drawer panel to allow users to switch among the views
3. Google Cloud service that handles the Google Cloud Messaging communication to listen to pushed messages from the servers and redirect them to appropriate components
4. Database services that provides method to access the logic models uses in the class

Each view of the application is implemented in a class called Fragment. Navigations between parallel fragments are achieved through Navigation Drawer as suggested by Google's new Material android design guidelines [11]. Each view is implemented as an independent fragment so that the user interface of the application can be refactored and reimplemented without changing the view itself. Transitioning between fragments is handled by the Main activity's fragment manager. The Main Activity also acts as the main communication among fragments fragments or between a service and fragments. The communication mechanism follows Observer pattern [12] with each fragment registers and unregisters itself with the activity for interested events.

4.3.2 Local database

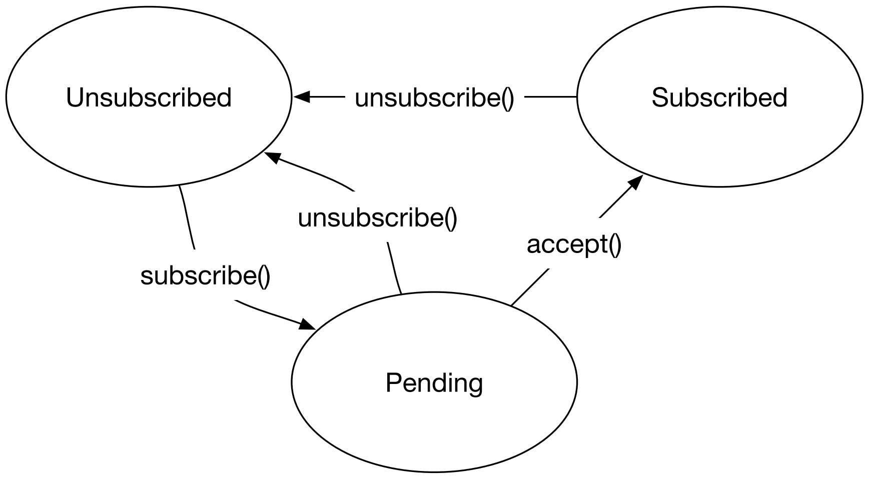
In order to interact with the local database, all views in the application use centralized point of access. The client application logic business requires 2 models `Data` and `User` and hence there are 2 contract classes that represent the interface to the database. These interfaces effectively simplify the dependencies and interaction between the views and the models. By having a single mean of access, it is easier to change the way the data is access globally by changing the interface and also make it easier to debug if necessary.

4.3.3 Utility Services

The application consists of a set of Service class extending Android's IntentService that handle interactions with the models and the application server. The service is invoked from any of the views by creating an Intent with appropriate arguments and the result is then propagated to all effective views that are interested in the result. Once finishing servicing a request, an event is created and broadcasted to the main activity. Since there are more and one view that are interested in a given model, the event is received by the main activity and then propagated to its current view for displaying the result if the view is registered with the activity. Upon creating and replacing a fragment to the activity, a fragment is registered to receive the broadcasted events following Observer pattern. Each event is implemented with a pair of Publisher-Listener interfaces with the view implementing the listener interface and the main activity implementing the according publisher interface. The service classes run on a separate thread rather than UI thread hence allows calling network operations in the service implementation. In this application, each request is translated to one or more network operations to the back end server and executed in an AsyncTask.

Subscription Service
The `SubscriptionService` is used to handle the interaction with the users and subscriptions. The service provides the methods to fetch a list of users of the application, subscribe to a user, cancel a subscription request and accept a subscription requests by other users. In local domain, the subscriptions from a logged in user to other users can be represented as a status attached to a User entity indicating not-subscribed, subscribed or pending request. The states of the subscription request is described in a Finite State Machine diagram. The aforementioned status is also implemented in the views to change the widgets accordingly so as to reflect the status of the subscription.

Figure 1: Subscription model FSM



Data Service
The `DataService` is used to handle the interaction with the data records in the database. In the client application, `DataService` consists of 1 method to fetch data from the back end server and store in local SQLite database. The fetching is performed in rolling basis to reduce the network traffic i.e. only the new data if fetched. Each subscribed user entry in the local database

maintains a `lastUpdated` field to indicate the latest time it was updated. This information is initialized to a zero day when the user is created and updated every time the data of that user is fetched from the database. Upon unsubscription, the entry is cleared and reassigned to zero day so that all data of that user is fetched if the user is subscribed to again. During the fetch and insert to database operation, the list of data entries is iterated and the latest date is recorded and updated for a given user.

4.3.4 Google Cloud Messaging

Google Cloud Messaging (GCM) is the free service provided by Google to allows developers to send messages between servers and client applications. The service supports both upstream and downstream messages but only downstream message is used in this application. To integrate the application with the Google Cloud Messaging service, each instance of the application must register itself and implement a listener service to listen for the pushed message [13]. Registration to the backend is invoked through a `RegistrationIntentService`. In the recent versions of Google Cloud Messaging library for Android, GCM register() has been deprecated and replaced with `InstanceID` hence InstanceID API is used in this application to keep it up to date with the latest change. Upon receiving the `InstanceID`, the service also invokes the registration to the application server (Endpoint component) to register its token which is used to identify the device. Receiving of downstream message is handled by subclasses of `GcmListenerService`. This is the receiving point that process the messages sent from the application server to signal changes of data or subscription e.g. request accepted events. The GCM listener service must be implemented so that it agrees on the format and content of the message sent by the server module.

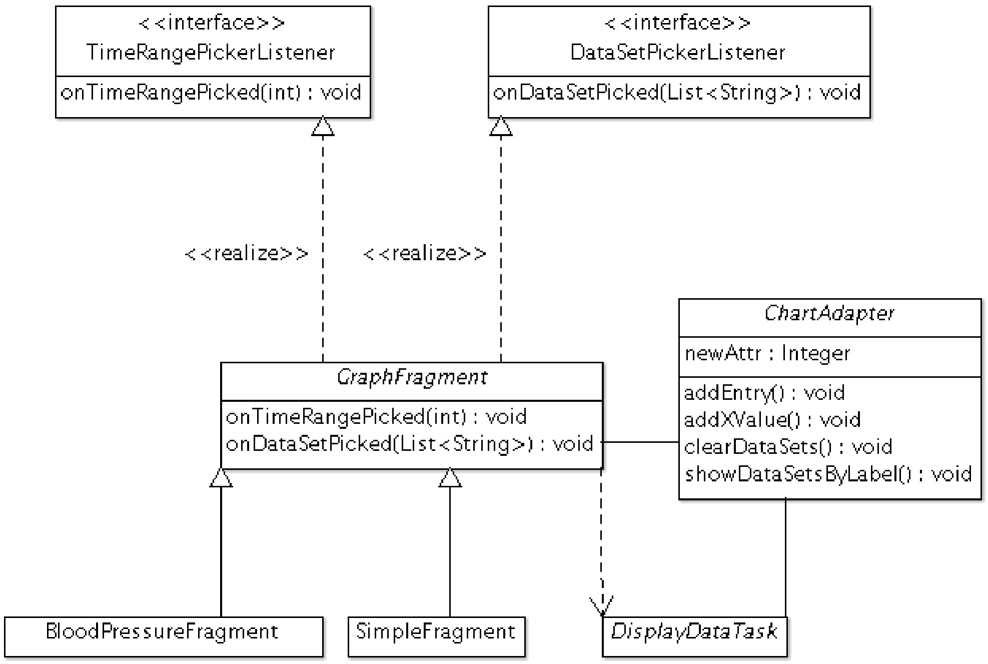
4.3.5 Graph display

Displaying of graph is contained in GraphFragment class. This fragment also allows users to choose various ways of representation of the chart such as which user, time range to display as well as appropriate style of chart for different types of data. Displaying of the widget is handled by MPAndroidChart by Philipp Jahoda [14], provided as an open source library on GitHub under Apache License, Version 2.0. The chart placeholder is populated with a Chart subclasses and then appropriate methods are called depending on the choice of representation.

Graph Fragment

GraphFragment and its subclasses are in charge of create the Chart and inflate it to the layout container. GraphFragment can be subclasses for specific type of data. In the scope of this project, it is implemented in 2 subclasses namely SimpleDataFragment that uses line chart with 1 entry per each x value and BloodPressureFragment that uses scatter chart with 2 entries per each x value. The subclasses are required to implement the abstract methods in GraphFragment to display the data accordingly. The GraphFragment contains the following abstract methods that are implemented in the concrete classes:

Figure 2: GraphFragment class diagram



```
abstract String getDescription();
abstract Cursor getQuery(HealthDroidDatabaseHelper);
abstract Chart makeChart(Context);
abstract DataPool makeDataPool();
```

With the appropriate dependencies constructed in the concrete classes, an asynchronous task extending from `DisplayDataTask` will then be invoked by on the time range selection and inject the dependencies dynamically. Actual invocation of the `Chart` widget is handled by a `ChartAdapter` object. However, each type of chart provided by MPAndroidChart has different attributes and hence is handled differently. The `GraphFragment` class only has dependency on abstract class `ChartAdapter` and relies on the concrete implementations to provide the correct way of interaction with the chart. Constructing of the `ChartAdapter` is declared as abstract in GraphManager and implemented by the its concrete subclasses, with each type of chart creating its suitable chart adapter. In the scope of this project implementation, the `Simple` creates its `LineChartAdapter` for inserting data to Line chart while the `BloodPressureFragment` creates its `ScatterChartAdapter` for inserting data to Scatter chart. Thus, when the application is extended for new type of graph, the new concrete class must create its own Chart adapter and changing to existing code is not required.

KeyCreator
The `KeyCreator` class has the responsibility of creating the values on x-axis as well as the lookup key in the resulted dictionary. The `KeyCreator` is declared as an interface and implemented by concrete classes namely `ByDayKeyCreator`, `ByWeekKeyCreator` and `ByMonthKeyCreator`

DataPool
As the graph fragment is required to display data grouped by date differently based on users' choices, the raw data entries acquired from database querying need to be processed and the accumulated value of a day, week or month is computed for displaying to the users. For each combination of choices provided by the users, the graph contains different number of entries on x-axis and the different corresponding y-value. To achieve that, every time the task is invoked, all data must be queried and prepared for processing in a `DataPool` object. The `DataPool` object acts as a central repository for data of all users in the local database to be grouped by their date and then creates a Map containing the grouped result. The `DataPool` class contains a mapping between a username and their relevant data entries as well

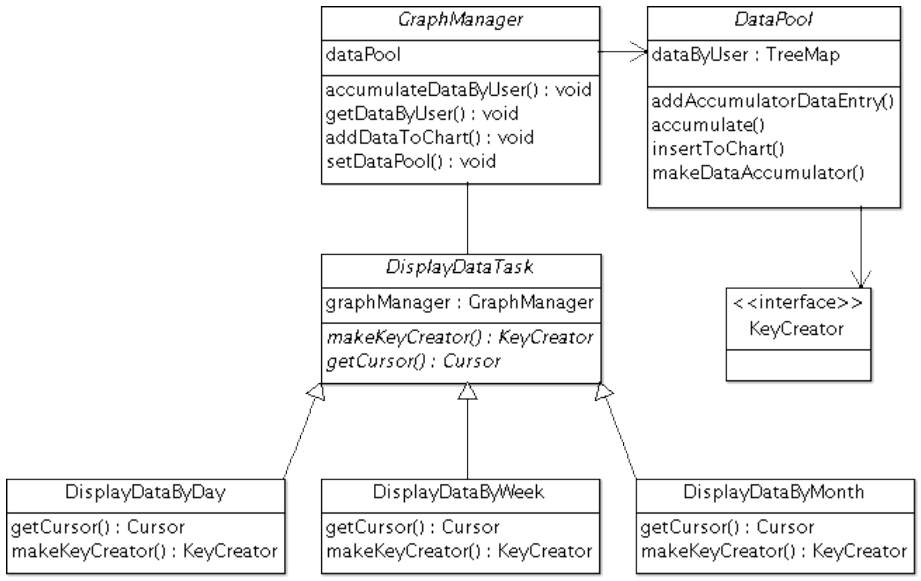
Method	Description
createKey()	this method takes a date string in RFC3339 format as parameter and returns the corresponding key string in the correct format
getDateRange()	this method takes the first date and the last date from the data set i.e. the range of values on x-axis of the chart and returns a wrapper object that contains these 2 values and normalize the range to be at least 10 units.
getTimeUnit()	this method takes no arguments and return the corresponding value from Java Calendar to help get the value from GregorianCalendar

Table 5: KeyCreator methods description

as the dictionaries mapping date to values. The resulted dictionaries contain appropriate keys generated by a KeyCreator object for each date range selection e.g. days (05/03/2015) for ”by day” choice and month (12/2015) for ”by month” choice.
`DataPool` depends on 2 interfaces namely `KeyCreator` and `DataAccumulator`. It can be injected with any subclass of `KeyCreator` depends on the date range required. Once invoked, the `DataPool` object iterates through all users in local database and creates a `DataAccumulator` instance for each user. Since there are different types of DataPool for each type of chart, the `DataPool` class is declared as abstract and requires its subclasses to construct the correct `DataAccumulator` instance.

DataAccumulator
The `DataAccumulator` class has the responsibility of containing a mapping from a string representing a date to a data representation. Depends on different type of data, the representation is constructed differently. In the scope of this project, DataAccumulator is extended by 2 concrete classes namely `SimpleDataAccumulator` that has 1 numerical value per date key and `BloodPressureDataAccumulator` that has 2 numerical values per date key.

Figure 3: DisplayDataTask



The `DataAccumulator` exposes the method `accumulate(String value, String key)` in which value is the String representation of the data and the concrete class is responsible for deserialize the string to achieve numerical values. `DataAccumulator DataAccumulator` has dependency on the `KeyCreator` class and required an instance of `KeyCreator` in its constructor. The caller of `DataAccumulator` class is required to provide the suitable `KeyCreator` instance for its need.

4.3.6 User Display

User RecyclerView

In recent versions of Android Support Library, Google has released the `RecyclerView` as a flexible view for providing a limited window into a large data set. This widget is a more advanced and flexible implementation of `ListView` that used to be the standard way to display a list of items. `RecyclerView` contains the view to display items that can be scrolled very efficiently by maintaining a limited number of views[15]. A recycler view delegate its responsibility to `LayoutManager` for displaying the layout and `Adapter` to map

Method	Description
accumulate()	this method starts the accumulation of data to create a map from user name to their data set contained in an object `DataAccumulator`
insertToChart()	this method takes a `ChartAdapter` as argument to insert the prepared data from accumulate() into a chart

Table 6: DataPool methods description

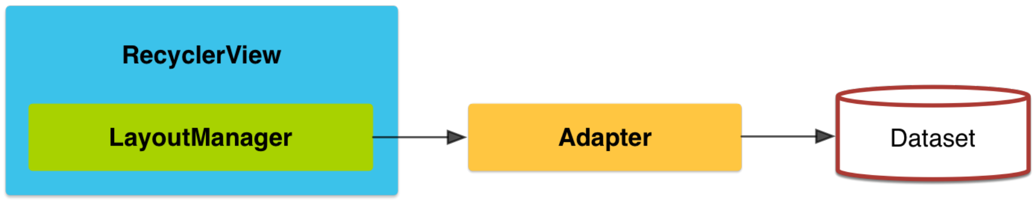


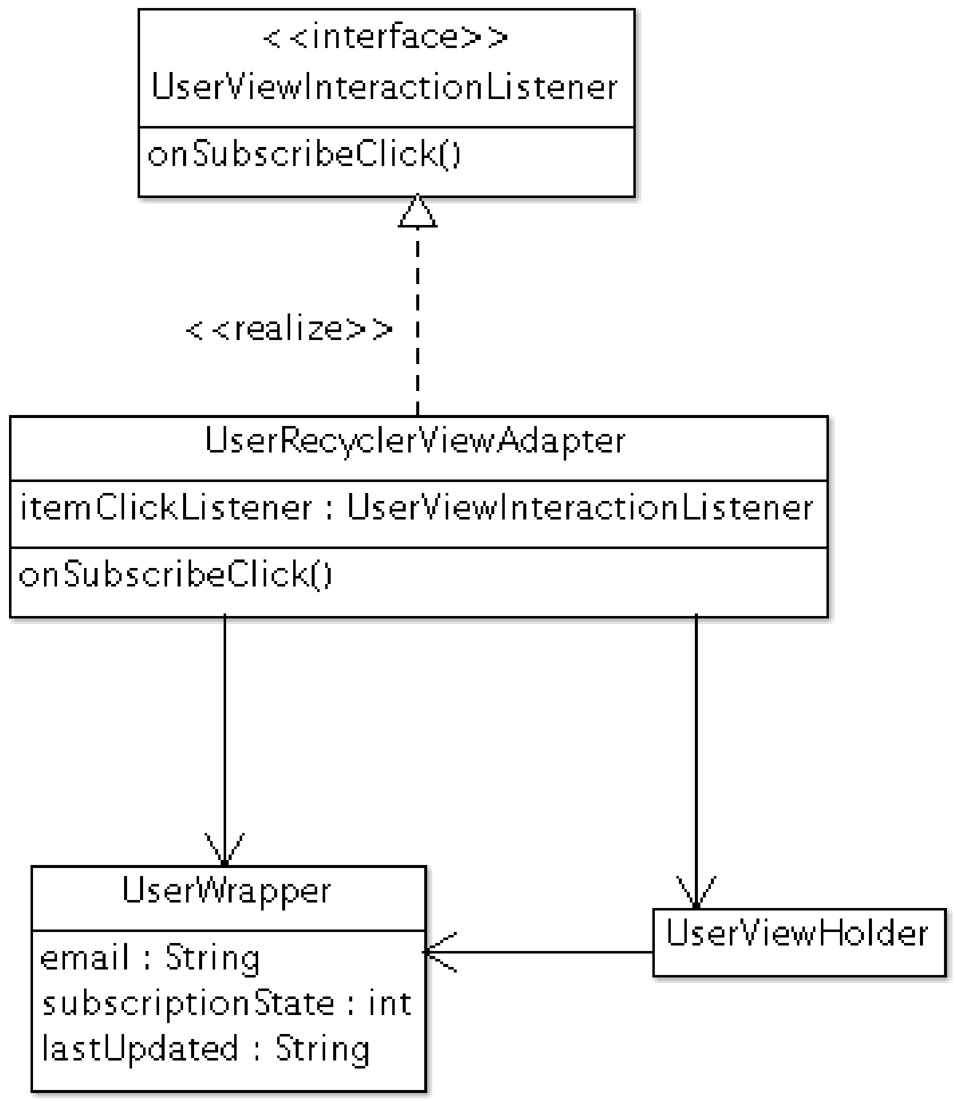
Figure 4: Recycler View structure

an object model to a view. When an adapter is created, the object representation is bound to the view contained in the corresponding `ViewHolder`. By delegating the responsibility to handle the binding and interaction with the views to the programmer, RecyclerView effectively reduces the overhead and unnecessary work in the UI.

User ViewHolder

The view for a user is managed by `UserViewHolder` class. This view contains the information about the users a button that allows to subscribe, unsubscribe depending on the current state. Interactions on the button is delegated to the event listener set by the client of this class i.e. UserFragment. Hence, interactions on the view is propagated upstream to the fragment to the database model and change of the model is listened by the fragment and propagated to the view to change accordingly. Thus, series of callbacks implemented by 2 interfaces `SubscriptionChangeListener` and `SubscriptionChangePublisher` created a 2-way binding of models in the application.

Figure 5: User view class diagram



5 Discussion

5.1 Android development practices

In order to approach the latest change in new Android version and Android SDK, the project aims to use the latest tools and practices suggested by Google. At the time this report was written, the libraries used by the application is updated the latest version available so that it can better keeps up with the trend and current trend. The project also uses the most up-to-date libraries and classes provided by Google in both Android application and backend server so that it has better performance and better chance for further development effort. For future work, the design and layout of the application maybe changed to suit Google's new design guideline as new version of android is released. Additionally, due to time constraint, only recent version of Android (from Android 5.0) is targeted and hence future work can be done to extend the application to support older versions.

5.2 Software engineering practices

The client side application was made with consideration about design and components choices. Different standards in software engineering was considered and followed throughout the development of the application so that the code can be read and reused by other developers. Serious consideration and effort has been put in the designing the graph components such that the application can be easily extended to other types of data i.e. blood pressure, blood glucose etc. as well as different ways of viewing data in the future. As the graph feature of the application is expected to be flexible and supports various options of displaying, the responsibility was split into small classes with single responsibility. Most components depend only on interfaces and hence new classes complying to the interface can be created and easily plugged into the code base with dependency injection. Otherwise, there would have been lots of code duplications in and type checking to ensure that the application works as expected. Without careful design, the code would have been tightly coupled and number of classes increase significantly as each class handles a specific combination of the choices. As the project proceeds, the unstructured code base could grow beyond maintainability.

6 Conclusion

In conclusion, this project has implemented a prototype of cloud-based medical data managing solutions. The application implemented in this application aims to provide a reliable platform for large volume of transactions. The application consists of an Android application and a backend server that runs on Google App Engine which is an efficient and scalable approach for server and database hosting. The projects provided working application as a proof of concept for a full end-to-end system that allows the users to post, retrieve and share their data. The work of this project also suggested a potential expansion to integrate with the real measuring devices so that recorded data can be submitted to the database in real time. Under careful consideration of the architecture and class design, the application can be easily extended to support more types of data as well as displaying options. Moreover, this project has targeted the latest libraries and android design guidelines by Google so that it can stay more up to date and maintainable in future work.

References

[1] X. Chao and F. Zhiyong, "A trusted affective model approach to proactive health monitoring system," in *Future BioMedical Information Engineering, 2008. FBIE '08. International Seminar on*, pp. 429–432, Dec 2008.

[2] G. F. Lawton, "Health monitor analysis system: Successful instrumented design and unexpected benefits," in *Autotestcon, 2006 IEEE*, pp. 677–682, Sept 2006.

[3] Y. l. Zhu, R. Li, X. b. Liu, and X. Jian, "Wireless communication technology in family health monitoring system," in *Business Management and Electronic Information (BMEI), 2011 International Conference on*, vol. 3, pp. 64–67, May 2011.

[4] I. Shabani, A. Kovaçi, and A. Dika, "Possibilities offered by google app engine for developing distributed applications using datastore," in *Computational Intelligence, Communication Systems and Networks (CICSyN), 2014 Sixth International Conference on*, pp. 113–118, May 2014.

[5] Y. Li, L. Guo, and Y. Guo, "Enabling health monitoring as a service in the cloud," in *Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on*, pp. 127–136, Dec 2014.

[6] J. S. Kim, B. O. Kim, and K. S. Park, "Development of hihm (home integrated health monitor) for ubiquitous home healthcare," in *Engineering in Medicine and Biology Society, 2007. EMBS 2007. 29th Annual International Conference of the IEEE*, pp. 363–365, Aug 2007.

[7] X. Jia, "Google cloud computing platform technology architecture and the impact of its cost," in *Software Engineering (WCSE), 2010 Second World Congress on*, vol. 2, pp. 17–20, Dec 2010.

[8] Y. S. Yilmaz, B. I. Aydin, and M. Demirbas, "Google cloud messaging (gcm): An evaluation," in *Global Communications Conference (GLOBECOM), 2014 IEEE*, pp. 2807–2812, Dec 2014.

[9] Google, "Cloud endpoints," 2016. [Online; accessed 7-March-2016].

[10] Google, "Search api," 2016. [Online; accessed 7-March-2016].

[11] Google, “Navigation drawer,” 2016. [Online; accessed 7-March-2016].

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[13] Google, “Cloud messaging,” 2016. [Online; accessed 7-March-2016].

[14] P. Jahoda, “MPAndroidChart,” 2016. [Online; accessed 7-March-2016].

[15] Google, “Recyclerview,” 2016. [Online; accessed 7-March-2016].