COMP 3004

D3: SYSTEM ARCHITECTURE AND DESIGN

Team: Divide And Conquer

App: Zippy - a grocery list service

Team members | Student Numbers:

Flavio Barinas	100799043
Geoffrey Duimovich	100975490
Jess Cannarozzo	101007447
Karla Martins Spuldaro	101021516

ARCHITECTURAL DESCRIPTION

APP Functionality Overview:

Zippy is an application which improves in-store grocery shopping experience in a flexible and convenient mobile form, designed for Android platform. It allows users to quickly and easily create grocery shopping lists, which can be shared among people with real time updates, while also providing them an interactive competition game experience.

Publish-Subscribe Style:

Firebase Database provides an API to subscribe to events for a specific path defined in the Database. The client can subscribe to a path and wait for onChildChange, onChildRemove, onChildMoved, or onChildAdded.

todo-items -> list_id: -> item_id author: String checked: Boolean item: string uid: String	owned -> user_key list_id: Boolean	shared -> user_key list_id: Boolean
todo-lists -> list_id -> access: user_key: Boolean author: String listName: String uid: String userCount:Integer	users: -> user_key: displayName: String email: String username: String	

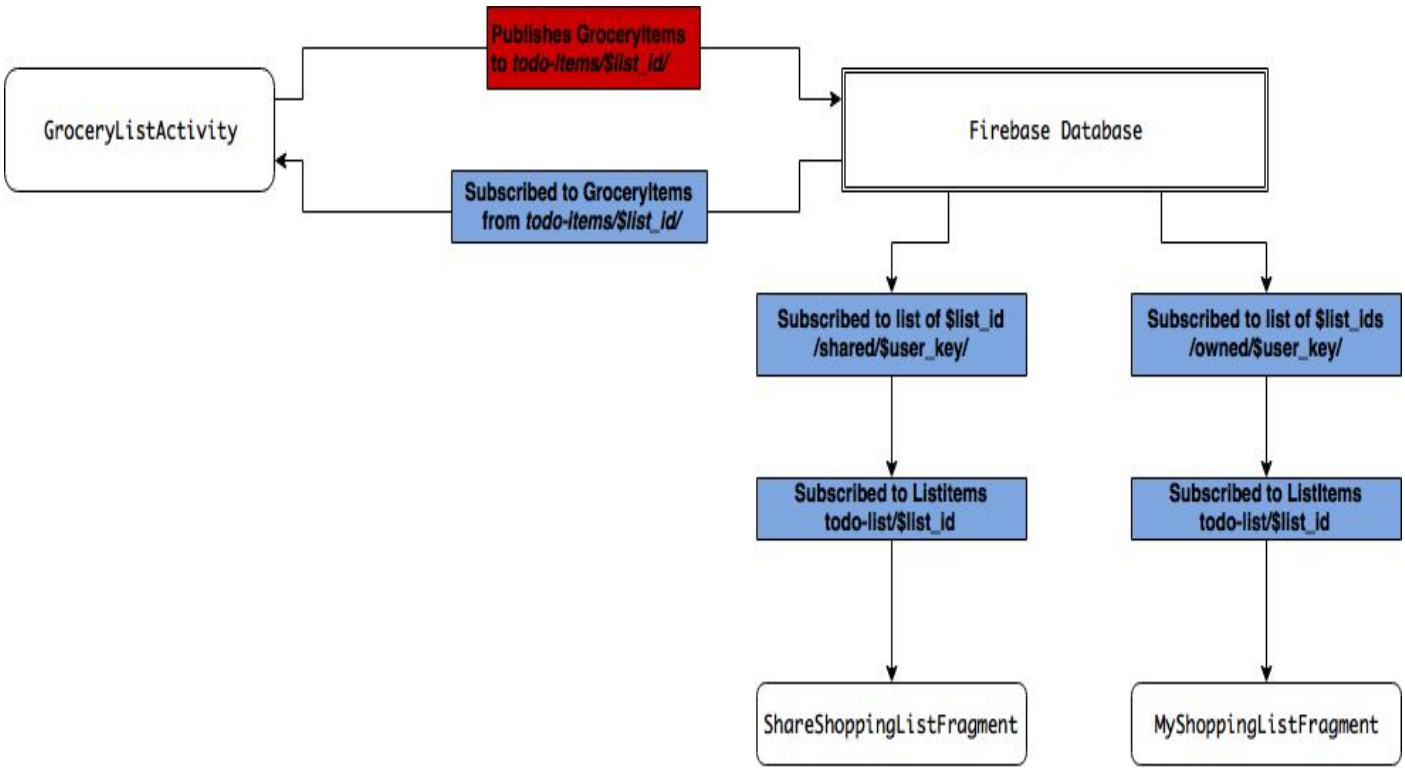
The choice of a publish subscribe architecture was chosen in order to fulfil the functional requirement of *Real-time List Editing with Others*. This is accomplished by having UI components such as Lists to subscribe to the data that it is displaying in the list from Firebase. When information is published to Firebase, all of the subscribing clients receive the updated data and the UI is updated.

A scenario this is used in is the GroceryListActivity class. When the Activity is created, it subscribes to the todo-items/$list_id/ path where all of the items in the shopping list are stored. When a GroceryItem is updated in Firebase, an event is received and the appropriate action, such as removing a GroceryItem from the list, causes the UI to update and have the item removed. If an item is changed, the UI will

redraw the new item that has been changed. When a user updates a UI element such as checking a box, an event is published to the Firebase database where the item that has been modified is updated. This creates an efficient data synchronization model between the clients since only the UI screen that is being displayed is receiving the updates.

Another functional property that the publish-subscribe architecture allows to be implemented easily is the ability to share lists with another user. When a user is waiting to be added to a list and they are in the ShareShoppingListFragment where the UI is being subscribed to a list of list_ids they have access to. When a new list_id is added, they subscribed to the /todo-lists/$list_id to process ListItems being changed, added, or removed.

Overall the choice of publish subscribe architecture choice is critical not only to support the functional requirements, but to ensure the non-functional requirement of performance. When a user navigates to a view that has a list, it will start subscribing to only the items in that list and will only update the elements that have been modified since it is subscribing to events instead of updating every single item even if one item is changed.



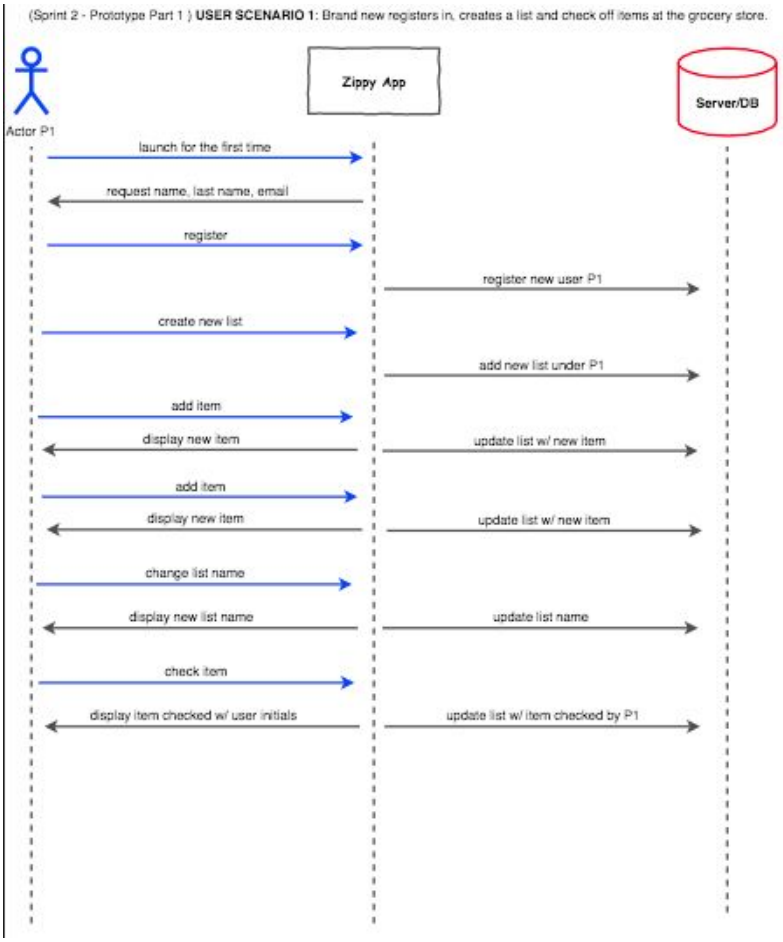
Event Based Style:

An event-based style was inevitable in creating Zippy, due to the way android applications are made and the nature of Mobile applications in general. Mobile applications almost entirely revolve around users interacting with a VIEW by pressing on the screen with their fingers, creating an event that will be dispatched to a listener for handling. In out app the interaction between the user and the UI(adding lists, adding elementing, switching between windows) is dependent on the event-based style. As such the there was no choice but you use the event-base style due to the development platform as well as the inherent user interaction of the app.

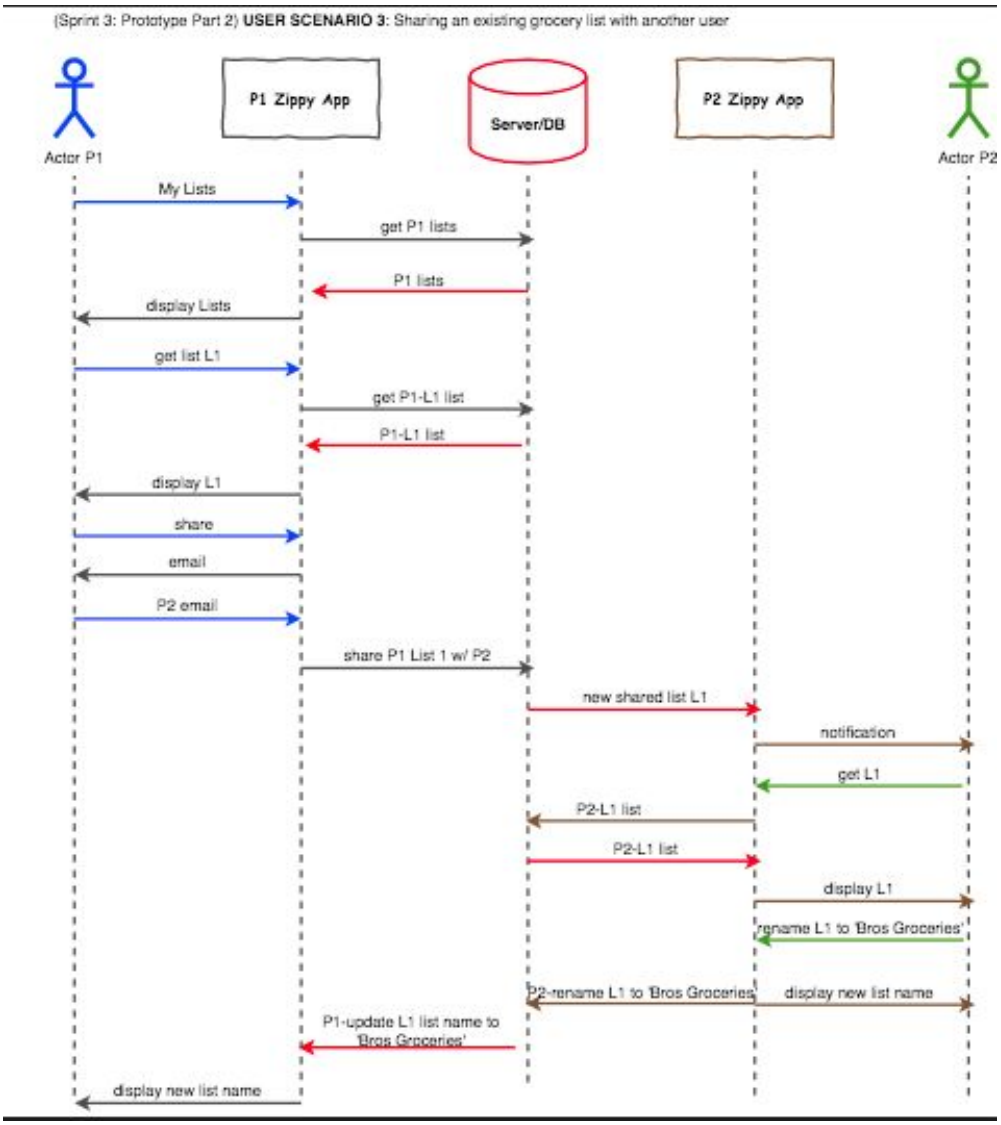
SYSTEM DESIGN

In the design process, both linear and cyclic strategies were followed, focusing on a top-down abstraction principle – first the main concepts were defined, then they are broken down into detailed tasks.

In the linear approach, first we defined what functionalities our system would have. Then we started a preliminary design by defining user scenarios and flow charts, in order to identify all the main components.

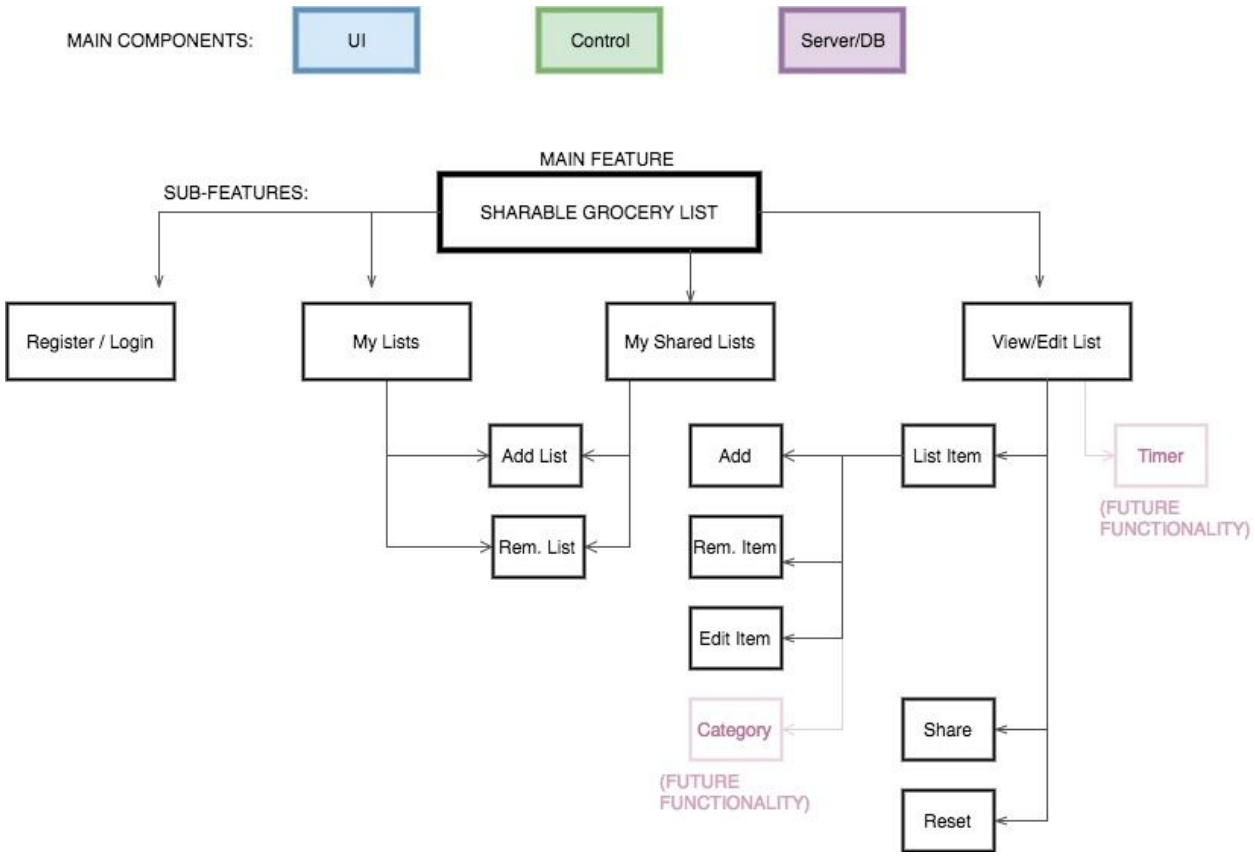


Preliminary user scenario 1 flowchart



Preliminary user scenario 2 flowchart

After selecting and learning more about the tools for implementing the application – Android Studio and Firebase – more detailed tasks for each component were then defined. Lastly, at the planning stage, milestones were created. Each milestone has one main goal (working product stage), and all the sub-tasks necessary for its accomplishment.



We followed Agile (Scrum) as a software development methodology. Since a new milestone (or Sprint) was due every two weeks, here we can also identify a cyclic strategy approach. At every Sprint, all tasks and milestones requirements were revisited, as well as the priorities of each sub-task.

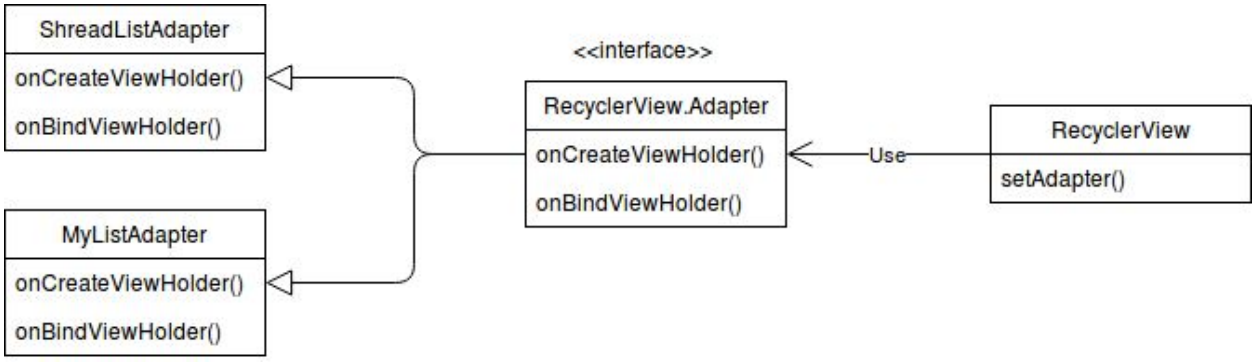
A month of planning and researching occurred before the development of Zippy officially began. Meetings were held once a week, sometimes twice, to discuss the next steps in designing the application. Colour schemes, placement of buttons, and the layout design of the lists all took place during the ten weeks of development. Amendments were made to the previous design to modernize its layout.

According to Professor Olga Baysal, a design pattern systematically names, explains, and evaluates an important and recurring design. The design patterns used in this project are described below.

Adapter Pattern

The adapter pattern was necessary in the implementation of Firebase. Creating an adapter to handle the loading of information (Firebase) and the displaying of information (UI), we were able to interface the partial loading with the UI view allowing for efficient querying and real-time updating taking full advantage of the publish-subscribe architectural pattern . Without the adapter, the communications between the app and Firebase would be slower and inefficient and would make the code difficult to follow.

The adapter pattern was selected over others because it was the most efficient way to ensure that data would be handled properly in an organized manner. An adapter allows for the moulding of any data in the exact form that fits with the UI. With an adapter, the application was able to be kept modular with its separate List class, Item class, User class, etc. The use of other structural design patterns would be unnecessary as objects only needed to be adapted in order to be properly displayed or have data be properly updated in the database. All of the objects within *Zippy* have different uses and were kept separate.

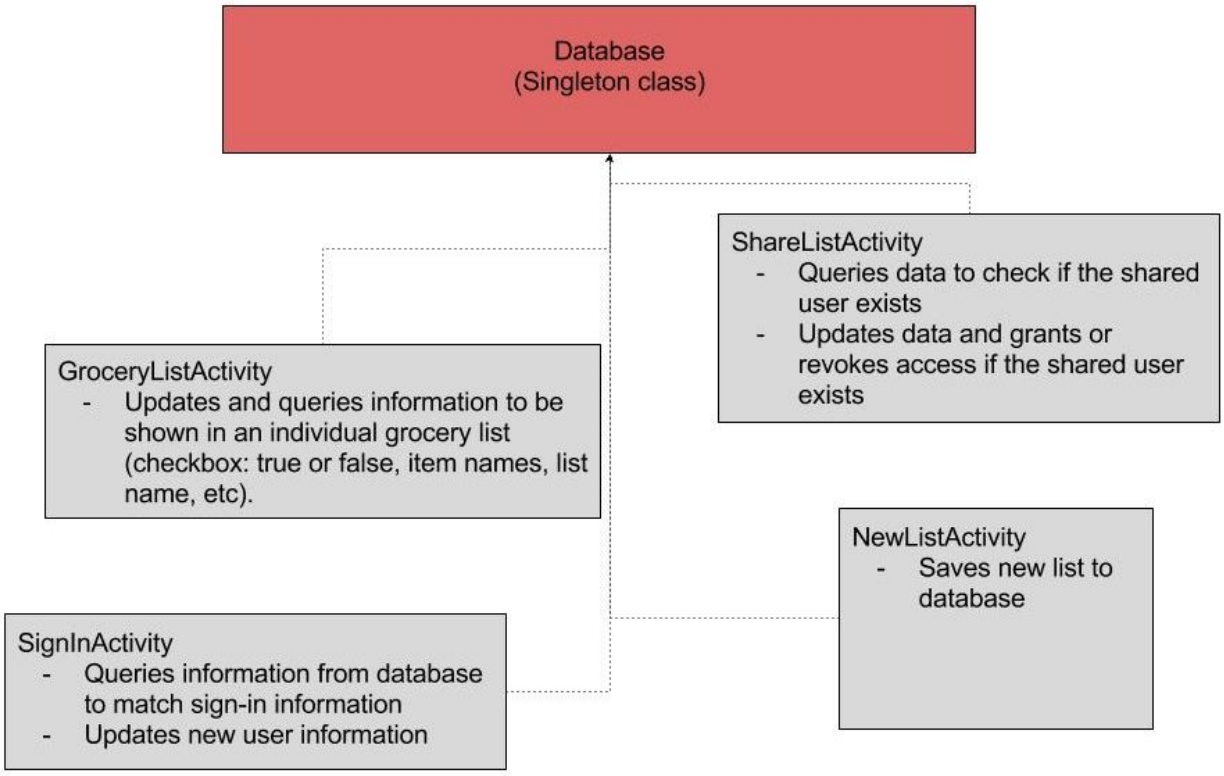


Observer Pattern:

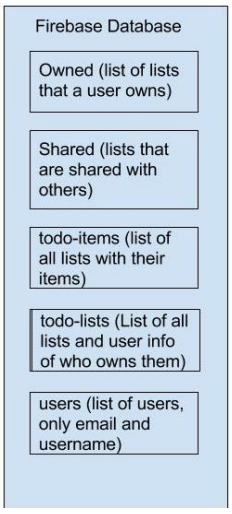
The Observer pattern is implemented in the ShoppingListFragment as an adapter and is used to support the Publish Subscribe architecture. The RecycleViewAdapter contains a list of of ListItem's, list ids and a list of ViewHolders which contain the UI elements for each cell in a List. When the Firebase event listener's fires an event, such as a changed event, a notifyItemChanged call notifies the RecycleViewAdapter that a viewHolder at a specific index has been changed. This allows for the subscribed events from Firebase to efficiently update the UI since they only notify the viewHolder objects that need to be updated instead of updating all of the items in the List.

The observer pattern is again used in the GroceryListAdapter class in order to support efficient UI updating to be able to have a UI that does not have noticeable latency. The event based architecture supports the observer pattern very well since events from Firebase listeners can notify the RecyclerView that objects need to be updated.

Singleton Pattern:



The database follows a Singleton design pattern because there is only one instance of it, and it can be accessible through any class. The above diagram is an accurate explanation of how our classes interact with the singleton database. Firebase allows for lazy initialization when needed, and from different reference points. In other words, our database is structured so that a class does not need to query the entire database in order to use one branch. For example, our database is structured as such in the diagram.



We selected this design over others because all of our classes manipulate data found in the database, so all the listeners needed to be accessible in case any change has been made to either a list itself, a list item, who has access to a list, etc. The majority of our classes also query data from the database in order to display everything in proper sequence for all active users. This order is important because the objective of our application is to allow users to make changes to each others' lists in real time.

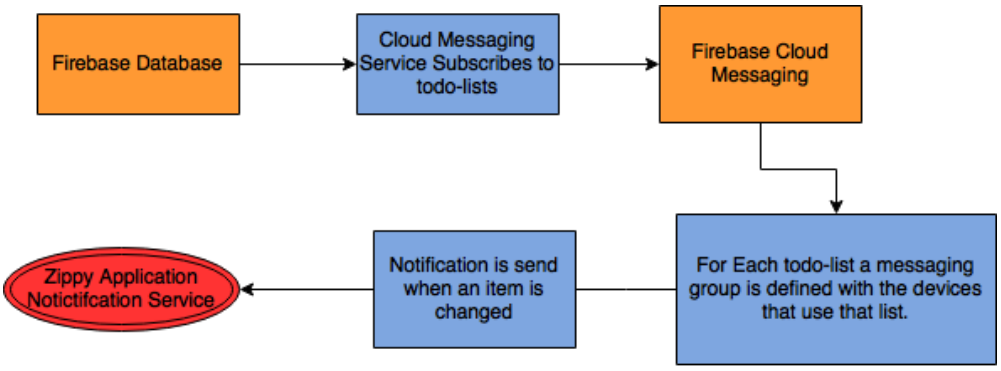
In the explanation for the Publish-Subscribe style, our database implements this by pushing data out to our client on any data change. Any user's individual lists subscribe to the data in the database, and data is updated as it is revised. This is so that users who have access to the list, but

who did not necessarily make the change, can also see it as it occurs. This could only be done through our single database, thus following a singleton design pattern as data is stored and queried from one source that is accessible from all classes.

System Alteration / Improvements:

An example of how our project could be changed is the addition of Android notifications. When a user has the app closed they do not receive any notifications currently about lists they are sharing with people. Currently a user would have to open up the app to be able to determine what items someone has checked off. Our current architecture depends on Firebase Database where updates are performed by the individual device clients. This means the app would have to be running in the background to be able to subscribe to events happening on the database. This is not feasible when the app is not working since the client would have to be subscribing to every item being updated for the user at once.

To be able to support notification's another Firebase component would be required to be added to architecture. This component is Firebase Cloud Messaging which allows devices to subscribe to messaging channels to receive updates about topics.



Messaging groups would be created for each list. When an item in the list is updated since the messaging service can subscribe to events in the database it can send a message to all of the device in the group indicating they have a notification. The Zippy Application Notification Service when running in the background displays a notification in the Android message tray.

TEAM MEMBERS TASKS

Over the course of ten weeks, all members equally developed and tested Zippy. Every task created included full end-to-end implementation. Therefore, every team member who was responsible for a certain task would be able to work on all system components. This approach was decided in order to give everyone the opportunity to better understand all application components and their integration. It was also a way for the team to effectively communicate and resolve issues quickly so that the development of Zippy could proceed.

Everyone has made changes to each others' features, whether to refactor or to add more functionality. Everyone also reviewed and approved all of the new feature updates before they were merged into the stable git branch.

Flavio

Flavio was responsible for the design of the Database and the application. As well as the interweaving of the UI and firebase updates(adapters), implementing the minor UI features and refactoring of the code.

Geoffrey

Geoffrey was responsible for the design of the application including implementing the functionality of the login screen, and creating scaffolding for the implementation of List View. Other stories he completed included removing a list.

Jess

Jess was mainly responsible for the design and implementation of sharing and unsharing.

- Helped to set up the event listeners for the checkboxes
- Added a couple UI features such a list name at the top of every list, and a way to view who has access to a list.

Karla

Karla was responsible for organizing the planning of the system, creating project milestones and tasks, based on decisions made by the group. She was also responsible for implementing some features, such as editing and deleting items from the list, and the reset functionality.

APPENDIX - Zippy Class Diagram