# Tuduz - A Task Organizer Platform for Class and Group Collaboration

Arizona State University - Barrett, The Honors College

*Project by Kelly Ndombe*

*Directed by Yinong Chen, PhD & Janaka Balasooriya, PhD*

# 0. Table of Contents

# 1. Abstract

There exist many very effective calendar platforms out there, from Google Calendar, to Microsoft's Outlook, and various implementations by other service providers. While all those services serve their purpose, they may be missing in the capacity to be easily portable for some, or the capacity to offer to the user a ranking of their various events and tasks in order of priority. This is that, while some of these services do offer reliable support for portability on smaller devices, it could be even more beneficial to the user to constantly have an idea of which calendar entry they should prioritize at a given point in time, based on the necessities of each entry and regardless of which entry occurs first on a chronologic line. Many of these capacities are missing in the technology currently used at ASU for course management. This project attempts to address this issue by providing a Software Application that offers to store a user's calendar events and present those events back to the user after arranging them by order of priority. The project makes use of technologies such as Fibrease, Angular and Android to make the service available through a web browser as well as an Android mobile client. We explore possible avenues of implementations to make the services of this platform accessible and usable through other existing platforms such as Blackboard or Canvas. We also consider ways to incorporate this software into the already existing workflow of other web platforms such as Google Calendar, Blackboard or Canvas, by allowing one platform to be aware of any item creation or update from the other platform, and thus removing the necessity of creating one calendar entry multiple times in different platforms.

# 2. Introduction

Like many other Universities, ASU is using services such as Blackboard and Canvas to manages classes, schedules and diverse other course-related items. This is beneficial to the school as well as the students given that all the information needed for a person's daily schedule exists and is stored through Blackboard or Canvas (class schedules, assignments due dates, appointments, etc.)

However, viewing all that information at once and at a glance is not always as straightforward as one may wish. Indeed, those platforms don't offer suitable alternatives for accessing the information for the smaller and more portable devices like smartphones.

What that implied for many students is the responsibility to write down into an electronic calendar --or some other form of planner-- the different entries that they would want to see on their portable device when briefly taking a look at it in an attempt to stay ahead of their tasks. The commitment to maintaining this exercise constant can easily turn into tedium and repeated work, especially when single events or tasks need updating every once in a while. That also means that not everyone will commit to keeping all their events and tasks in one place. While this practice is not necessary for a person's success or productivity, having all of one's calendar entries organized in one place can certainly make a difference in their productivity. This is where the idea for this project comes in.

The tedium aspect in the manual organization of such calendars is what started the conversation of having an automated system handling a user's calendar entries in the most effortless way for the user themselves. A system that would determine priorities and a scale of urgency from the events of a given user, and provide an adequate visual of those events, to equip the user with a perfect sense of what needs their attention at a specific moment, at all times. But most importantly, a system that could be accessed conveniently from a mobile device like a mobile phone. In addition, a system that would aggregate various information pertaining to a person's calendar (from their ASU account as well as various other event platforms such as Google Calendar and Outlook) into one single view.

# 3. Objectives

The primary goal of this project was not so much to generate new data as much as it was to organize the data that is already out there into something more meaningful, more directly recognizable and utilizable by the users. Therefore, the objectives of this project are (1) to organize users events in a way to provide them with better insights on their calendar entries and the necessary chain of actions at a given time for their best productivity; (2) to make the services of this platform conveniently accessible through a mobile device (3) to find a way to aggregate information pertaining to a user's calendar from various platforms (ASU, Blackboard, Google Calendar, etc.) in one single place. All that, without pulling the users away from the different routines they are already used to. This means that the users must be free to stick with any pattern of actions they are accustomed to when it comes to creating their calendar entries, and still be provided with the services of the automated system. Whether they are used to creating assignments through Blackboard or making their tasks through Google Calendar, the user should not, as much as possible, have to produce the double work of reentering the same entry into a second platform to benefit from the organized event-suggestion.

# 4. Requirements

Following some design discussions, it was decided that we would put an emphasis on the transferability of the data from the platform used for this project to any external platform.

This mostly means that we had to carefully choose which platform we were going to be using to store user's data. Some providers can be easier in the scenario that we need to export data, and some may be a little more intricate for that simple of a task.

Therefore, the first approached as discussed with the committee, was to go about this project using an XML database. However, it became apparent, later on, that a different tool could be used for better productivity. Indeed, as the work went on with the use of an XML database (and a C# base for all web services as well as desktop client), the comparison was made between the use of such technology and the use of other technologies such as those offer through Google's Firebase. After additional review, it was decided that the project would completely switch from using XML and C#, to using Firebase and Angular.

In addition, one requirement that we've tried to observe throughout the design and implementation process was to stick as much as possible with the native tools of any particular technology we were using. That means that we tried to minimize and avoid using any external libraries or independent packages to not tie the project to any potential fluctuations of those external tools. If anything is needed that is not offered through the native tools yet, as much as possible, we try to create what is necessary to not depend on any external sources.

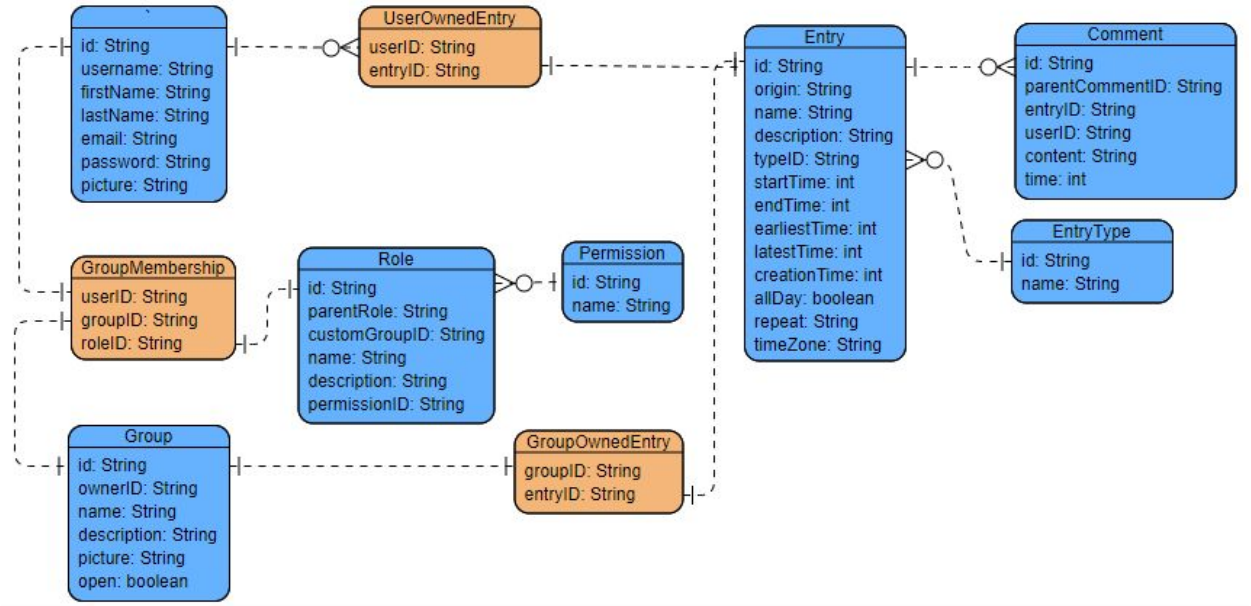
# 5. Procedure

## 5.1. Plan

To approach the realization of this project, the first step was to design an abstraction of the structure that the whole system should have.

The most work would eventually go into figuring out and designing an appropriate configuration for the database supporting the system. This is that we needed to first figure out what were the key aspects of this project and how they all connected together.

The following is the first attempt at a database diagram representing the different entities that would be needed and only marking the connection between entities that should be aware of each other.

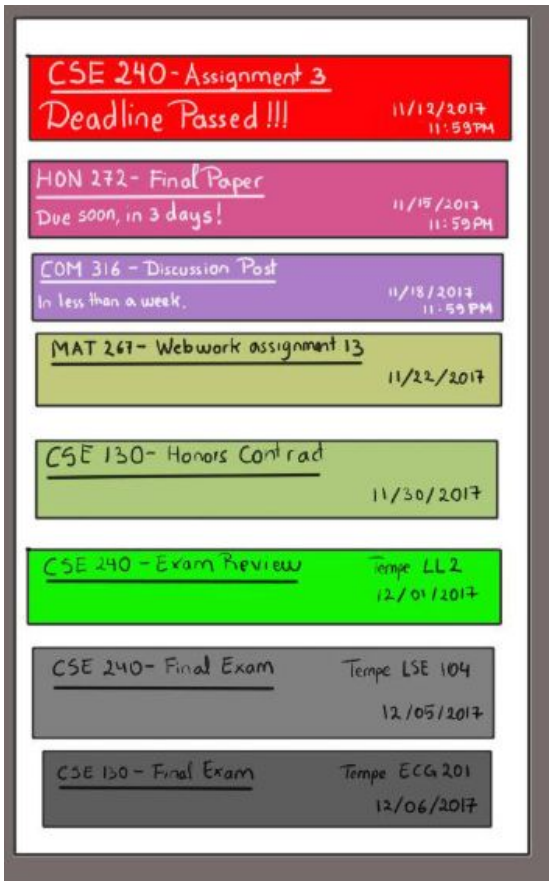



Next, we would have to actually build the backend and database system to support the platform. However, as it was revealed later, this representation played in our disfavor when it came time to implement the database. This because, at the time of coming up with this diagram, we had in mind that the project would be making use of a relationship-supporting database such as SQL or MySQL. However, given that we have later pivoted to using a NoSQL database, a lot of rethinking had to happen to make the relationships possible.

Once the backend and database ready, we would have needed to build a web client frontend to communicate with the backend and database, and correctly display the information in a way the users would make sense of it. This client should allow the users to perform all CRUD (create, read, update, delete) actions on any necessary data as appropriate.

After that, the task would have been to similarly build a mobile client that would replicate the behavior implemented in the web client. Because of time constraints and because of a much bigger background in Android programming than in iOS available, only one mobile client should be prioritized, the Android client.

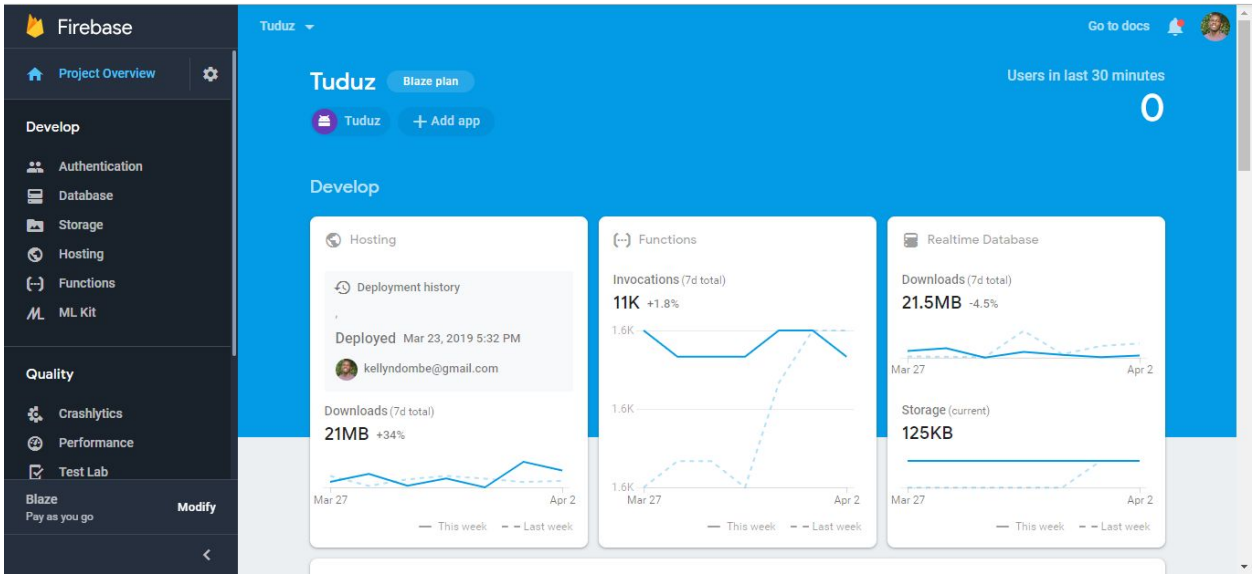
Here is an early Mockup for the intended design for priority ordered entries.




## 5.2. Technology Used

- Github (free, student's private repositories): For version control and bug tracking.
- Firebase Database (free): As database
- Firebase Bolt (free): Since the database at Firebase is a NoSQL database, it uses "rules" to allow us to impose validation criteria on the kind of data that is written onto the database and who can access that data.
- Targeryen: For Firebase rules testing
- Firebase Authentication (free): For any user authentication, password handling and security.
- Firebase Hosting (free): For hosting the entire website
- Firebase Functions (free for 12 months, then charged by function invocation): For any web services needed for the app such as cron-jobs, cleaning jobs, etc.
- Angular 6: For the web client
- Android Studio: For the Android mobile client

# 6. Database & Backend - Firebase




## 6.1. Implementation

### 6.1.1. Database

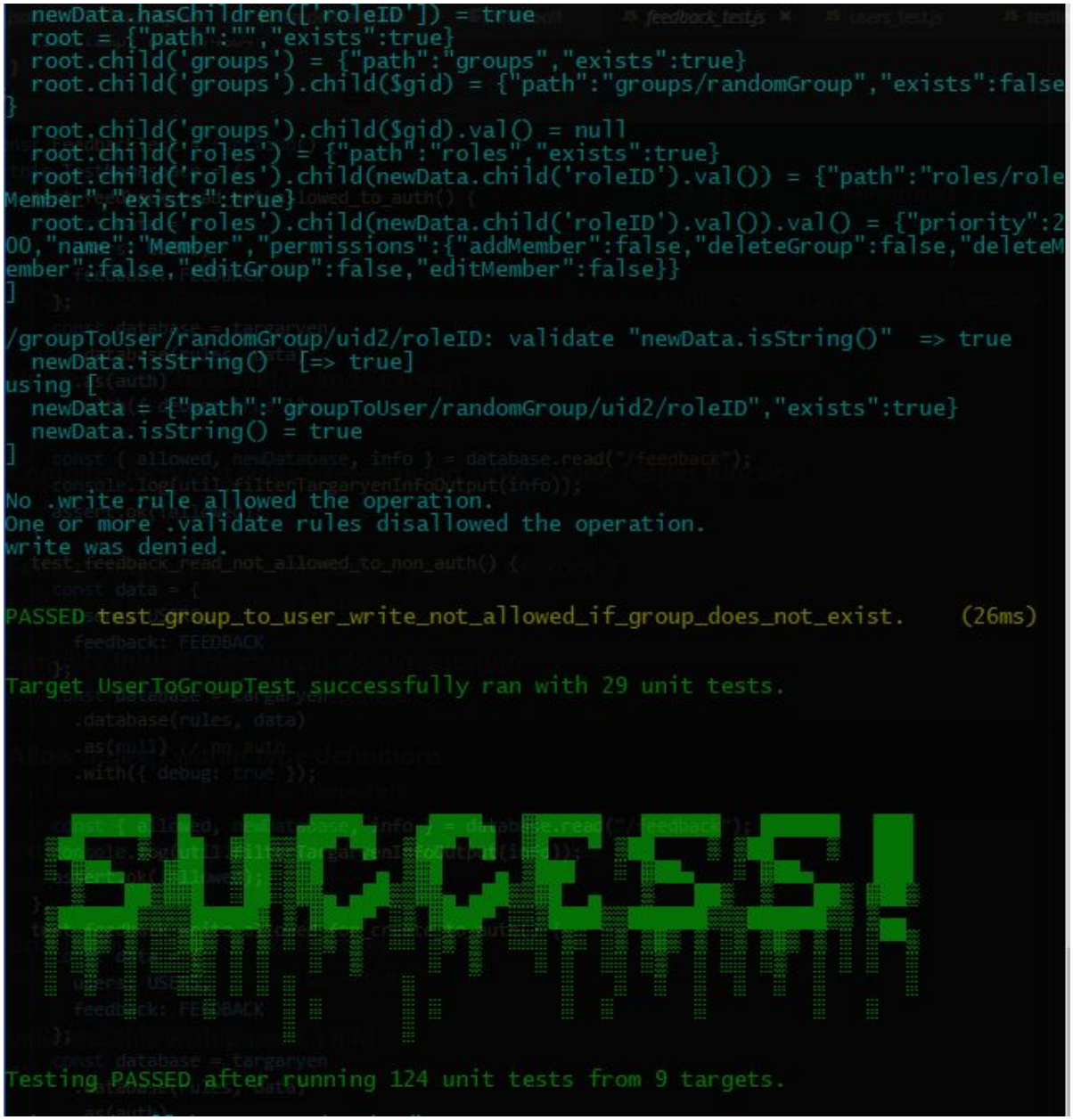
Firebase was used for the database and backend of the system. More specifically, 2 projects were created. One serving for development purposes, used to implement experimental features; the other serving for production purposes, containing all the confirmed features.

The database itself is pretty straightforward to use. It is a NoSQL database, which means it may need to be structured in quite a special way in order to accommodate the various relationships used in storing the data. Indeed, the security and integrity of the data is ensured thanks to Firebase Rules. These rules are written in the form of a JSON file and contain information indicating specific constraints for any data that should be allowed into the database. Those rules also offer to customize the security aspect of the database by specifying constraints around user allowed actions. This is that it allows specifying what kind of user can view/modify any specific kind of data.

Editing the rules from the JSON file is relatively easy for smaller requirements. When there is a need to add more constraints into the mix, it becomes less advisable to do it directly in the JSON file as it can be hard to maintain or even update over time. For that reason, we made use of Firebase Bolt, a language offer a friendlier alternative for writing lengthier and more

complex rules for a Firebase Database. The language is experimental and open-source. Many contributors are still working on making the language more reliable and fit the purpose of use of the many developers using it. Nonetheless, it remains a powerful tool allowing a much better productivity.

In addition, to add on another layer of safety for the security of the database, we incorporated unit-testing for each document (a.k.a tables) of the database. For this, we had to create a custom-made testing framework in order to allow for a more customized unit-testing experience as well as a better logging of any particular reason for failure. Indeed, with Firebase Rules, an unexpected behavior will not usually provide much explanation for the error other than “PERMISSION DENIED”. We then had to use the services offered through Targeryen (Goldibex) (an offline Firebase testing library) to simulate Firebase operations and get a more verbose logging. This was not as easily achievable using existing Node JS testing frameworks such as Mocha or Chai. That’s why we had to come up with a custom-made framework allowing us to separate each unit from one another and get verbose logging for failed tests. Here is a screenshot of a successful test run, containing all the needed logging information.




As for the implementation of the testing framework itself, it is somewhat similar to the known structure in other frameworks and makes use of the assertions just as much. Here is a snippet of the testing for the “feedback” section of the database containing rules indicating who can read/write feedback into the database, and the structure of this feedback.

```
25  const FEEDBACK = {
26    fid1: {
27      title: "Just spreading love",
28      description: "Just wanted to show some appreciation <3",
29      client: "android",
30      timestamp: 8349574884
31    }
32  };
33
34  const FeedbackTest = function() {
35    this.TestNamespace = {
36      test_feedback_read_not_allowed_to_auth() {
37        const data = {
38          users: USERS,
39          feedback: FEEDBACK
40        };
41        const database = targaryen
42          .database(rules, data)
43          .as(auth)
44          .with({ debug: true });
45
46        const { allowed, newDatabase, info } = database.read("/feedback");
47        console.log(util.filterTargaryenInfoOutput(info));
48        assert.ok(!allowed);
49      },
50      test_feedback_read_not_allowed_to_non_auth() {
51        const data = {
```

The tests for different documents (a.k.a tables) are separated into different JavaScript files and can all be run at once by using the “npm test” command.

### 6.1.2. Authentication

One big advantage of using Firebase was that it comes with its own authentication system right out of the box. This means that it offers a system security and encryption handled by Google itself. Firebase gives the choice between different methods of authentication, from which the developer can use the ones that they want to support. Thanks to this flexibility, it is relatively easy to incorporate sign in/sign up using an email and password, a Google account, or event accounts from Facebook, Twitter, Github, etc.

### 6.1.3. Hosting

Firebase also offers the possibility to host any static website. This option is particularly advantageous when building a website using some of the popular frameworks known today such as Angular, React or Vue. Indeed, Firebase offers the options host any form of static content as long as the entry file is named “index.html”.

The content will be available at a Firebase hosted URL. However, it is possible to redirect any domain named to point to this content, as long as ownership of the domain name can be proven. This is what we have used for this project to redirect the content generated through Angular to the URL https://www.tuduz.com.

Moreover, by using this service, we are ensured an SSL certificate, as can be seen from the URL. See appendix for a tutorial on how to set up Firebase hosting.

### 6.1.4. Functions

The backend of the system in this project necessitates that certain regular tasks run in the backend when needed. Indeed, the system relies on the fact that the entries should have a "priority" component that would determine their importance at a given time. This priority component is meant to be very dynamic and should ideally be updated every minute since it can happen that the order of priority between 2 entries changes over time (depending on the selected urgency when creating the entry and the current time). This computation is mainly done in the frontend when the entries are loaded into the client. However, to ensure a better experience for end-users, it is necessary to also have a form of priority in the backend so that the client would know which entries to load first (since not all entries are loaded at the same time, but rather they come by batches). For this to be effective, the entries need to be updated in the backend regularly so that their "server priority" reflect as closely as possible what their actual priority would be when calculated by the frontend.

For this purpose, there is a function running in the backend fulfilling the role of an hourly cron-job. The chronologic invocations were implemented using the App Engine Cron (Haskins). In addition to this cronjob, there other functions (internal web services) allowing for some cleaning routines. Such jobs include the handling of any deletions and any cascading that needs to happen (handled manually since the possibility of constraints offered through SQL-like databases is not available); or the creation of repeating entries.

These functions, unlike the other services above, will only be free for a period of 12 months. After that period, the account will be charged a certain amount per function invocation. See appendix for a tutorial on how to set up a Firebase function.

## 6.2. Challenges & Limitations

### 6.2.1. Many-to-Many Relationships

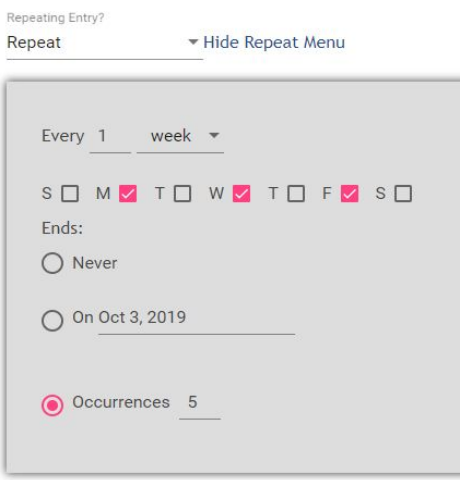
As it was mentioned above, one of the errors made during the design phase of this project was to think about the database in terms of a relational database whereas the actual implementation was to be done in a NoSQL database in the end.

This surfaced mainly when it was time to handle what is known as “many-to-many relationships”. A database such as that offered through Firebase is nothing but a JSON file. For this reason, it does not offer possibilities to perform operations such as connect data from different tables by defined constraints. Instead, the developer is encouraged to design the database structure in such a way that any querying will be made easier, but most importantly feasible.

The downside here is that we have had to adopt design policies that would otherwise be inadvisable due to the unchecked dependency between different entities that they generate. Indeed, we were faced with the inability to ensure constraints between some data (ensure that the deletion of a data cascades into deleting a depending other data) and the need to restructure the database in a way that would make querying many-to-many relationships possible. This meant, unfortunately, that we have to be creating duplicates for some of the stored data. The danger of such a design is that new operations may be added with the omission of the fact that duplicates are featured in the system. Thankfully, this is made a little safer by introducing Firebase Rules that ensure that operations on one data necessitate a given set of requirements having to do with other duplicate data.

### 6.2.2. Repeating Entries

Finding a design suitable for the handling of recurring entries necessitated some thinking. Indeed, there are many ways that one could use in order to efficiently store repeating events in a calendar (P). This is the case when a user wishes to create an event that they want to repeat with a certain pattern, for a given amount of time.




Two approaches were possible when a user would indicate that they want a certain entry to be repeating.

We could either decide to only store the one created entry with a defined “recurrence pattern” that we would use every time we’d need to display the entry. This approach, while beneficial for memory use, may lead to some serious performance issues on the frontend clients. Indeed, the necessary computation from the store “pattern” can easily become very heavy on the client for any task that is repeating along a relatively long period of time. In such a case, depending on how wide the interval of recurrence is, the client side computation may have to perform a costly computation (costly to the flow of the UI).

The other alternative was to create all the necessary occurrences in the database when a user sends the request. The downside of this approach is that in some cases, it will necessitate a certain memory capacity if the repeating feature came to be overused. We have, however, preferred to go with this second approach as we want to minimize the amount of computation for each client, to ensure the best user experience.

In order to correctly implement this approach then, we had to slightly modify the design of the object structure corresponding to the objects storing each entry. Initially, each entry is attributed a unique ID at creation, in addition to all the other necessary fields for storing a typical entry. To accommodate for recurring entries, we needed to add a “recurring ID” field. If 2 or more entries are sharing the same recurring ID, then they are part of the same entry recurrence. That means that when a user requests the creation of a repeating entry, the backend will make sure to not only create the said entry, but will also make sure to give this entry a recurrence ID and generate new entries that will carry the same recurrence ID. This recurrence ID will be helpful later when the user requests any sort of edit on one of the occurrences that should also affect some of the other occurrences of the same entry.

These additional operations are made possible thanks to Firebase Functions. Indeed, we are using some functions that are triggered once every time an entry is created, updated or deleted. They make sure that all the occurrences of an entry recurrence stay consistent according to the last user request. Therefore, a user request, as far as entry recurrences are concerned, contains a field called, the “onServerUpdate”. This field contains 2 elements. An “action” and a “target”. See the beside figure which is a screenshot of Firebase Rules pertaining to the server update feature. The action takes one of the values create/update/delete. The target can be all/from/only. Those help the server determine if the newly created/updated entry should impact any of the other entries sharing the same recurrence, and how.

```
68        // restrict the values of onServerUpdate.action and onServerUpdate.target
69        (this.onServerUpdate == null ||
70        ((this.onServerUpdate.action == 'create' ||
71          this.onServerUpdate.action == 'update' ||
72          this.onServerUpdate.action == 'delete') &&
73        (this.onServerUpdate.target == 'all' ||
74        this.onServerUpdate.target == 'only' ||
75        this.onServerUpdate.target == 'others' ||
76        this.onServerUpdate.target == 'from'))) &&
```

### 6.2.3. Database Lookup with Special Characters

Firebase has some configurations that make it impossible to look up a certain set of characters. Indeed, one of the actions one can perform with Firebase is query the database by filtering with a certain option. This is useful when looking up, for instance, entries with a name matching a certain given sequence of characters. Unfortunately, this is not possible if the sequence of characters contains any of the following characters

- . (period)
- $ (dollar sign)
- [ (left square bracket)
- ] (right square bracket)
- # (pound sign)
- / (forward slash)

This made it impossible to perform operations like looking a user by their email address, because the period(s) contained in the address made it impossible for the query to be successful. One way to remedy this was to rethink how we were storing the emails. While the email associated with the authenticated user would not change, the alternative was to store, in addition, a slightly edited version of the email address with all the periods replaced by commas. So the address “example@gmail.com” of an authenticated user would be stored as “example@gmail,com” in the database.

### 6.2.4. Creation Time

We also experienced a few difficulties with the handling of the current time with respect to the Firebase Database internal clock. Indeed, in the initial design, we had plans to store each entry with a field marking the entry’s creation time. With that, there was the need to incorporate a rule to make sure that the written “creation time” was indeed in the past (compared to the Firebase internal clock, at the time of the rule check). This check is necessary in the event that we want to keep this field for each entry. The field, itself, would be used in determining what entries need to be archived at a certain point in time, when the entry is old enough (to ensure an optimal performance for data lookup).

This works well if the client making the creation request to the database has a clock in sync with that of the Firebase Database. However, in the very rare event that a client device would be slightly ahead of time (for whatever reason), the created entry would have a creation time slightly ahead of the current time in the internal clock of the Firebase Database. This

means that the creation would never be validated as the “creation time” would constantly be invalid. We’ve had to put this feature aside to focus on other more important matters, in the hope of revisiting a different implementation in the future.

# 7. Web Client

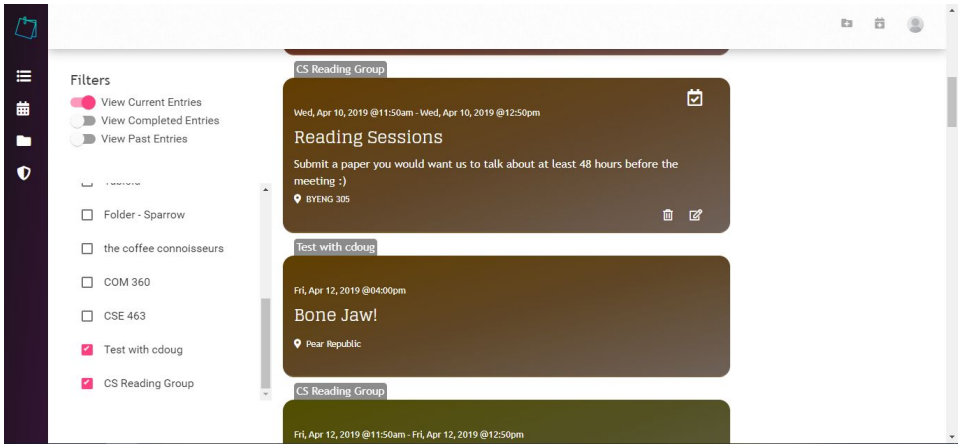
The web client is available at https://www.tuduz.com

## 7.1. Implementation

The web client was created entirely using Google's Angular 6. In addition, Google's Material Design was extensively made use of to help with the styling and feel of the Web App. Other popular tools such as CSS have also been used to allow some animations.

The Web App has been divided into 3 main sections:

### 7.1.1. Priority List



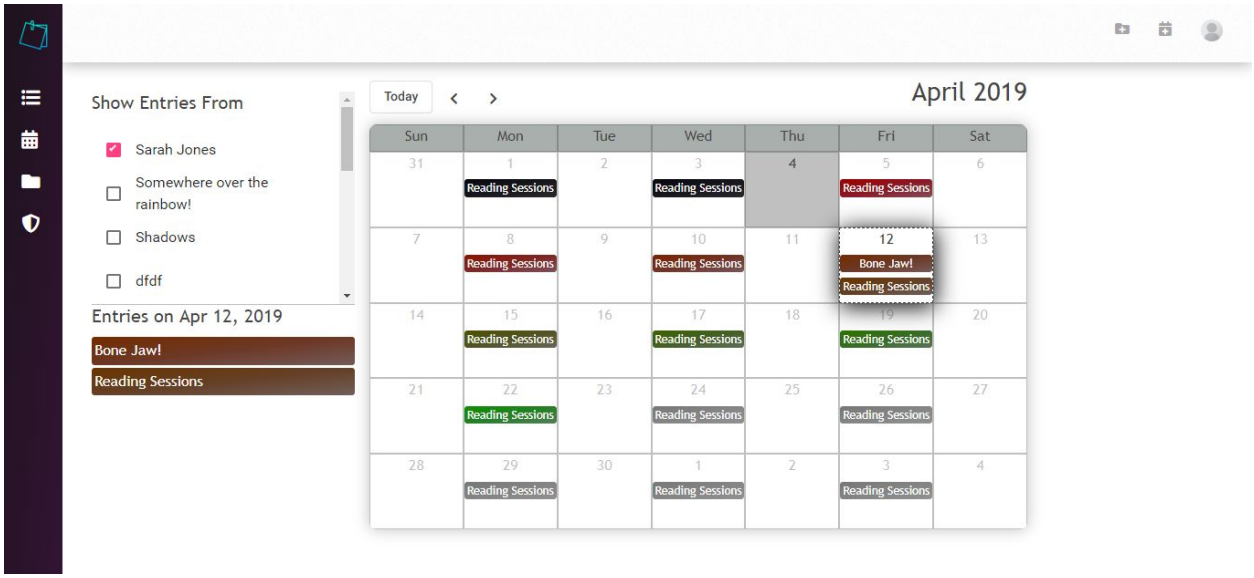

The priority list section displays to the user their entries based on their assigned priorities. This necessitates that the entries be assigned priorities when they are downloaded from the database. This priority is calculated based on two fields contained in each entry: ***earliestTime*** and ***latestTime***. Those fields each store a date & time converted in milliseconds and are set when the entry is being created. They respectively correspond the *earliest time that the said entry should become relevant for anybody to care about* (meaning that it's somewhat useless to care about this entry before the specified time) and the *latest time the entry should be dealt with* (meaning that it's recommended that this entry should be dealt with before the specified time).

For example, a professor could create an entry corresponding to an assignment for a given class. The earliest time field here could be given the date and time when the assignment will be made available to the class. The latest time field could store the latest recommended time to have finished the assignment.

Based on those two times, the client will compute a priority by checking where on the scale defined by those two fields the current time stands. The priority will normally range from 0% to 100%, except if the entry is not yet relevant (negative value) or if the entry's completion time has already past (greater than 100%). Once the priority has been defined, each entry will be displayed with a color corresponding to its priority. In addition, if the entry is still active and is getting more and more urgent, a message will be displayed on top of the corresponding entry card to let the user know of the amount of time left before the entry expires.

A user has the possibility to mark a certain entry as "Done" at any point, by clicking on the checkmark at the top right of the said entry.

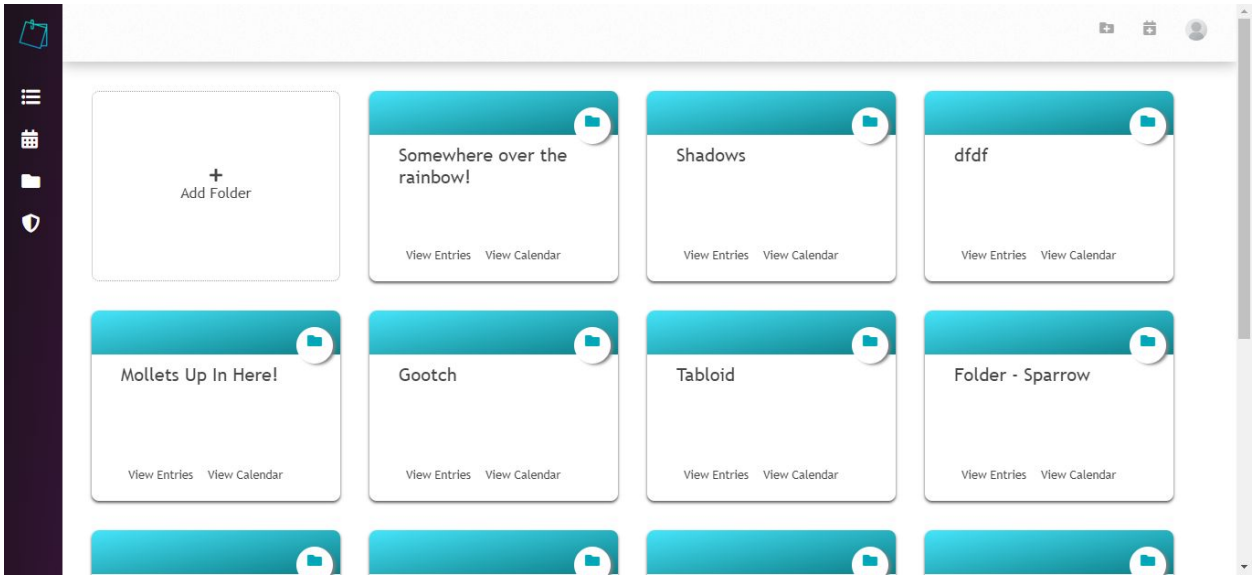
### 7.1.2. Calendar




The calendar section offers to the user to see their entries on a calendar layout. For this, the frontend requests from the database to get all entries with a "start time" contained in the current calendar window. Once the entries are downloaded, they will still be assigned a priority and a corresponding color.

Most of the display is similar to that in the priority list section. Clicking on individual entries will still bring up a dialog window containing the information about the said entry.

### 7.1.3. Folders




One of the purposes of this system is to allow better collaboration in various group settings. It does so by offering to manage the group's events and tasks at once, meaning that one person could be responsible for adding, editing entries relating to any given group, and the remaining members of the group would simply need to open their App to be updated. This is not an uncommon feature as many calendar service providers have a similar implementation (either with event sharing or group creation). The advantage here is that the group members get to benefit from the prioritization order without needing to, themselves, figure out how to order different entries.

A folder has one or more admins who have total power of action within that folder. That means they have the ability to add/delete members, give/remove admin rights, change the information of the folder, or delete the folder.

A folder can be public or non-public. A public folder offers a link that anybody can use to add themselves to the folder and see its content. A non-public folder is only visible by people who have been added by an admin of the said folder.

## 7.2. Challenges

### 7.2.1. Infinite Scroll

One issue that we faced was allowing infinite scroll in the priority list section. Indeed, it is safe to assume that the user may have a relatively long list of entries to display. In such a case,

it is not a good design to ask the frontend client to download all the required entries at once. That would impact the performance of the device and would offer a bad user experience.

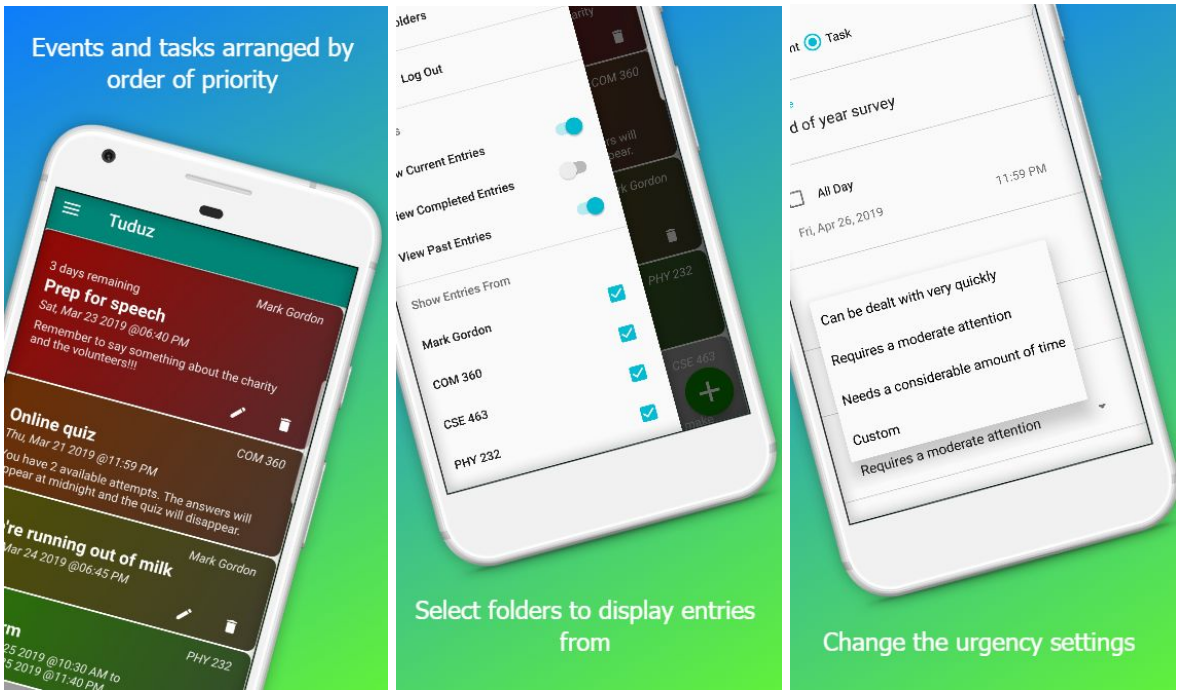
Instead, there are a few alternatives to avoiding such performance issues. One could either use pagination, such as the one used on any regular Google search page, or infinite scroll, like seen on most social media platforms such as Twitter or Facebook. We chose to go with the latter to allow for continuous content.

As mentioned in the requirements, one of our concerns was to generate anything we needed internally if it didn't exist yet, instead of adding external dependencies. This meant that we had to generate a way of achieving infinite scrolling since it is not natively offered through Angular as of this writing.

Although we did not use external libraries, we were inspired by some of the techniques used in some implementations out there (Orizens). In our implementation, we watch the value of the scroll bar on each scroll event. Every time this value indicates that the page has been scrolled more than a certain threshold (85%), we notify to the process that is listening, which will, in turn, request more data and update the page. As this update is happening, it is necessary to use a throttle value (500 milliseconds) in order to not trigger the event multiple times for one detection.

However, later changes were brought to our listening pattern of the RealTime Database that forced us to temporarily put a pause on the use of infinite scroll for the web client. Instead of the content loading automatically when the user nearly reaches the bottom of the scroll, we have added a functionality for the user to indicate that they want to see more data, by clicking the corresponding button at the bottom of the scroll.

# 8. Android Mobile Client

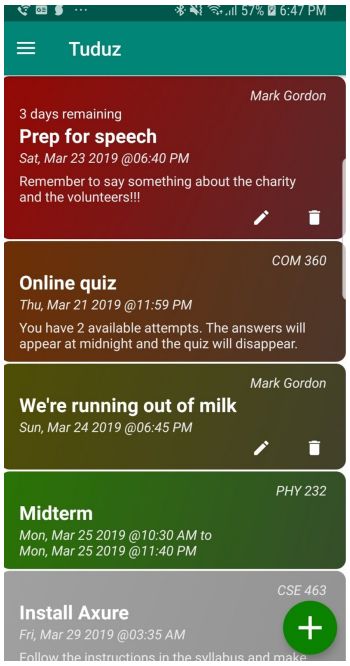



The Android Client, at the time of writing, is available for Beta testing at https://play.google.com/apps/testing/com.tuduz.android

## 8.1. Implementation

The Android Client was created using Android Studio. In addition to the native packages, Google's Material Design elements were used for ensuring a consistency in theming.

### 8.1.1. Priority List

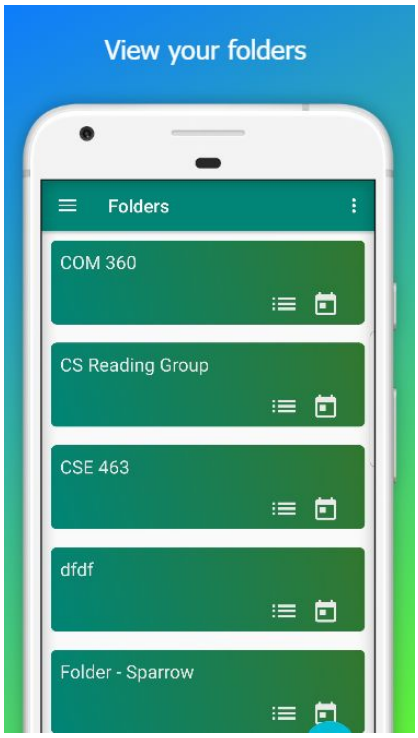



The implementation of the priority list follows the same logic as in 7.1.1. The lists have been built using Android's well known Recycler Views. The Recycler View is using a custom made ItemTouchHelper to support the swipe motions (swipe left-to-right or right-to-left). The Cards are also making use of transition labels to help in switching between different activities by allowing a smooth transition (eg: clicking on an entry to view it will give the impression that the entry simply expanded to fill the whole screen, while in reality a new activity has started).

### 8.1.2. Calendar

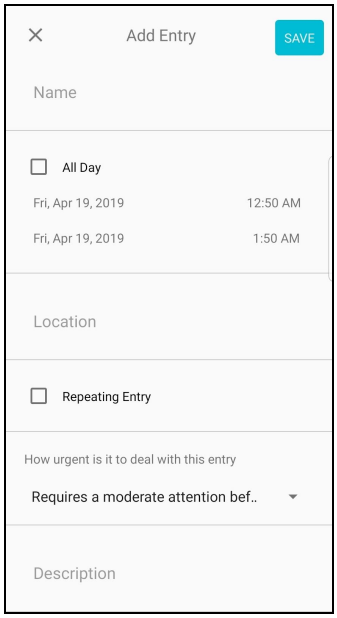

The implementation of this calendar follows the same logic as in 7.1.2.
There was a need to create a custom calendar as the Android's default "CalendarView" would not provide the flexibility to edit the different cells. This custom calendar has been built using LinearLayouts and GrridAdapters. Each cell has a corresponding date (with a color depending on the current month) and up to 3 entries (the 3 most prioritized of each day) respecting the priority color-coding.
Clicking on a month cell that has at least one entry will automatically scroll the view down and reveal a scrollable list of all the entries on the selected day.

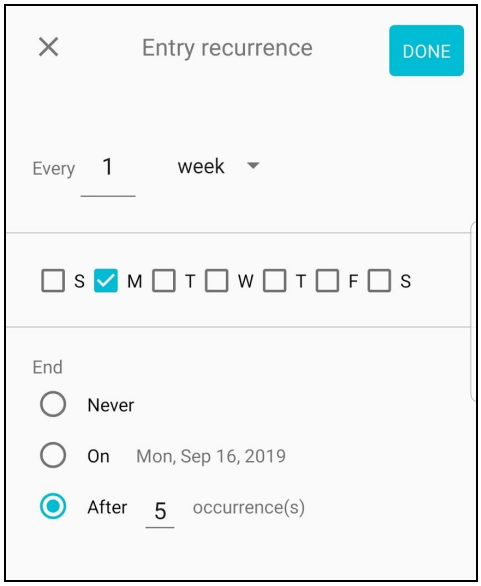
### 8.1.3. Folders




The implementation of this folder section is again following the same logic as in 7.1.3. These are again implemented using RecyclerViews.

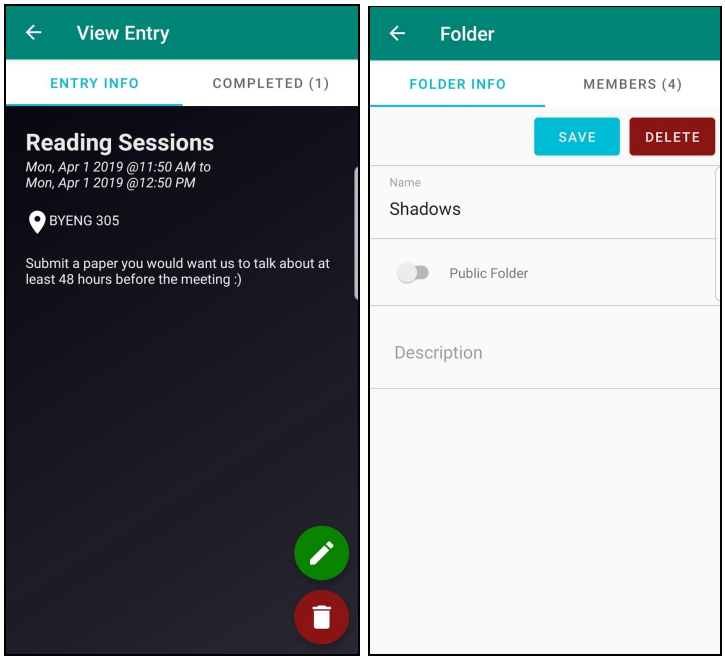
### 8.1.4. Create Entries




The entry creation menu has been created with a model very close to that used in the Google Calendar.

### 8.1.5. Tabs & TabLayout

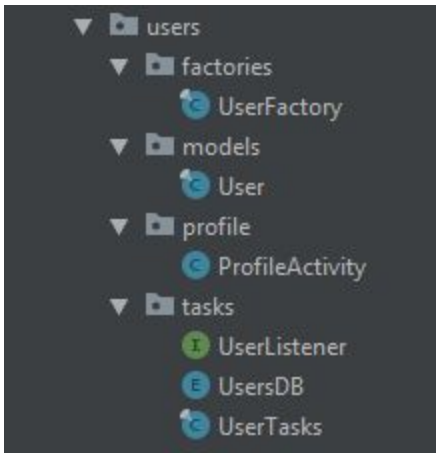



We are using Tabs in order to combine certain groups of views. Like shown above, the activity allowing to view an entry contains 2 tabs: one with the summary of the said entry, and the other with the list of users that have already completed this entry if it's shared. Similarly, the activity to view a folder's information contains 2 tabs: one with the folder's info and the other with a list of the members.

### 8.1.6. Architecture

The project is divided into packages corresponding to particular modules or most likely corresponding to object models used in the App. The main object models used here are *entry, calendar, feedback, folder, username,* and *user*. There are other packages corresponding to other modules (a "common" package with sub-packages containing custom pre-conditions, annotations, calendar implementation, date-time class, etc.)

For the sake of simplicity, let's focus on the "users" package.




In each such package, there are at least 3 sub-packages:

- factories: containing any factories needed to create or manipulate models under this package
- models: containing any models used in the App.
- tasks: this sub-package contains all the necessary code to handle the asynchronous work related to this package. The class *<model>Tasks* (here UserTasks) is the only one communicating directly with the database to request asynchronous data (all crud operations are performed in here). That means that no other class is allowed/supposed to request info from the database (excluding the implementation of onChildEvents in some parts of the UI). Any of that asynchronous work as to be requested through this class. The *<models>DB* (here UsersDB) is a package-view *enum* that defines constants for the properties for this model, persistent with the database.

```
enum UsersDB {
    USERS("users"),
    FIRST_NAME("firstName"),
    LAST_NAME("lastName"),
    USERNAME("username"),
    EMAIL("email"),
    PICTURE_URL("pictureURL");

    private String label;
    UsersDB(@NotNull String label) { this.label = label; }

    @Override
    @NonNull
    public String toString() { return label; }
}
```

This is only used when the "task" class needs to perform operations or request information from the remote database (hence, package-view). That class makes use of the values returned from these enum constants to identify the desired fields (instead of hard-coding them)

In addition to these 3 packages, there are other packages corresponding to the UI implementation of the particular model. In this case, we have a "profile" package with the implementation of the "Profile" activity.

## 8.2. Challenges & Limitations

### 8.2.1. Firebase Android & Class Export

Firebase has the advantage that it is thoroughly documented and has a very present support community. This is just as true for the Android client of Firebase as it is for Firebase in general. However, some of the [allegedly] intended behaviors have raised some issues during our development. Indeed, some functionalities of Firebase did not always match what we may have conceived them to be at first glance.

Such is the case of class export. Indeed, Firebase offers the possibility to simply export Java Classes into the database directly in order to generate in the database an object with the same data members as the exported class. However, we have quickly learned that the “data members” are not exclusively attributes and can also be generated from some of the class’ methods. Indeed, methods that appear to be getter methods may well be interpreted as referencing an implicit data member, and Firebase will therefore store a property of that name. We remedied the issue by ensuring that we are keeping track of the classes that should be exported and those that shouldn’t, with the use of custom made annotations.

The “Exportable” annotation is used to indicate that a certain class is intended to be exported into Firebase and therefore should not contain any methods other than getters and setters for the intended data members.

```
1   package com.tuduz.android.common.annotations;
2
3   import ...
7
8   /**
9    * The <code>Exportable</code> annotation is used to indicate that a class represents
10   * a model that can be exported to the Database.
11   * Using a Java class to set an object on the Database with <code>ref.put(class)</code> can result
12   * in exporting an object with more fields than expected. Indeed the presence of some additional
13   * methods (getters or setters) may be interpreted as a field by Firebase.
14   * This annotation is to avoid such confusion by ensuring that the said class does not contain any
15   * methods other than the setters/getters for the desired fields.
16   */
17
18  @Retention(RetentionPolicy.RUNTIME)
19  @Target(ElementType.TYPE)
20  public @interface Exportable {
21  }
```

Here is an example of the “User” class bearing the “Exportable annotation.

```
1   package com.tuduz.android.users.models;
2
3   import com.tuduz.android.common.annotations.Exportable;
4
5   @Exportable
6   public final class User {
```

In addition, we encountered what we thought to be an unexpected behavior with the Firebase “addedChild” event. Indeed, in our understanding of the event, it should notify its subscriber only when a new node has been added on the Firebase database. However, according to the design specifications of Firebase, this event is not only triggered each time a

new node is added, it also triggers for each already existing nodes at the time the event object is being created. This means that when the App would start, the portion of code detecting that a new node has been added would be fired for each one of the nodes currently present in the database, then again when new nodes are actually added. This forced us to rethink our implementation of the different subscriptions.

# 9. Distribution

There are possibilities of using the Tuduz platform as provider for other clients to integrate into their own systems. This is that we are considering making an API for this platform so that other clients (like ASU) could easily integrate their event tracking system with Tuduz. They would do that through the API interface, by using the provided method calls to perform any necessary actions.

# 10. Improvement

One of the things we were not able to achieve during the timeframe of this project was the addition of a browser extension feature to incorporate this App into the use of other Calendar providers. Indeed, the idea was to use a browser extension that would the App to detect when a user would be creating or updating a calendar event in a given calendar service provider (like Google Calendar, or through Blackboard or Canvas). By detecting such actions, the extension would then be able to either create or update an entry on behalf of the corresponding Tuduz account. One of the objectives of this project was to keep the experience as uniform as possible for users that are used to other similar services. One way we achieved that was by staying as close as possible to the standard user experience (especially for the interfaces involved in creating new entries). However, this was intended to allow the users to keep creating/updating their calendar entries through their usual platforms, while at the same time benefiting from the priority ordering of Tuduz. It also serves the purpose of eliminating the need for the user to create duplicate events (one in their normal platform, and another one in Tuduz)

Another feature that did not make the cut was the voice command incorporation. Indeed, we had plans to add to the mobile software the possibility to perform all the actions using a voice command. The initial suggestion was to use an existing voice command provider like Alexa (mostly to benefit from the existing collaboration between ASU and Amazon). But we have also considered manually generating an NLP model for a better customization of our commands, actions and responses.

Moreover, one of the software engineering processes that we have not been able to make extensive use of was testing. Indeed, per advisable SWE practices, we would have ideally wanted to generate unit testing for each new module created or updated. This is particularly made easy by tools such as Android Studio. In addition, given the Android background of the development team working on this project, it was easily feasible to ensure consistency between the unit testing and the actual implementation.

However, given the few modifications our specification underwent and the fact that we've had to pivot our backend (and rebuild it from the ground up), we have chosen to prioritize the testing only for the Database side since we want to ensure that no harm would happen to the internal system. This is that we wanted to prioritize the testing for the database (See 6.1.1) by ensuring that we have control over what is being written into and accessed from it. The extensive unit testing that we have implemented serves to confirm that our assumptions on the safety (based on the defined Firebase Rules) are indeed correct. In addition, since we approached the project in a quite incremental way, it was necessary to make sure that any new additions were not breaking or interfering with any of the previously working behaviors.

Therefore, one of the primary tasks moving forward would be to go through each module in the Android and Angular implementation and generate unit tests for each one of those.

In addition, the current version of the platform is only accessible given a certain number of criteria. Indeed, the mobile version only exists for Android at the moment; the web App is recommended on browsers like Google Chrome or Firefox and will not necessarily correctly display the content in other browsers.

# 11. Conclusion

This project has been a great opportunity to put in practice the different principles of Software Design and Engineering (and a part of Architecture) and experiment with different approaches and design patterns in order to find an optimal policy. While the final product shows some potential for commercialization, one prominent outcome here is the experience that has ensued from the manufacture of an end-to-end, multiplatform application.

Although the number of platforms reached at the moment is limited, we still consider it to be a good thing that we have reached the current stage. One of the next steps would be to diversify the reach of the software to more platforms like iOS, Windows Phones, BlackBerry, etc. Similarly, the current web implementation will optimally work in a Google Chrome or FireFox browser. The future efforts would be in diversifying the compatibility to other browsers.

# 12. References

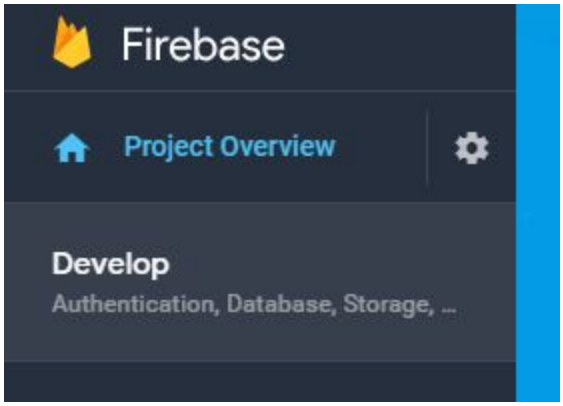

- Goldibex. (2018, May 13). Goldibex/targaryen. Retrieved April 3, 2019, from https://github.com/goldibex/targaryen
- Haskins, A. (2017, March 24). How to Schedule (Cron) Jobs with Cloud Functions for Firebase. Retrieved April 3, 2019, from https://firebase.googleblog.com/2017/03/how-to-schedule-cron-jobs-with-cloud.html
- Orizens. (n.d.). Ngx-infinite-scroll. Retrieved April 4, 2019, from https://www.npmjs.com/package/ngx-infinite-scroll
- P, T. (2015, December 17). Design and manage recurring events. Retrieved April 03, 2019, from https://medium.com/@ScullWM/design-and-manage-recurring-events-fb43676e711a

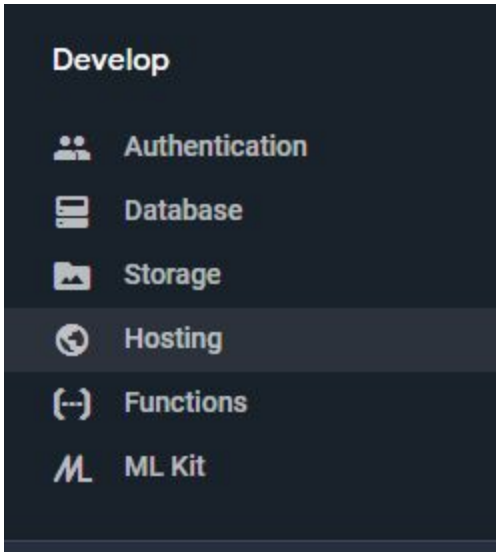
# A. Appendix - Tutorials

To follow these tutorials, you will need a Gmail account and Nodejs installed on your computer and available through your command line.

Here are the steps to set up a Firebase project. See (A.1) or (A.2) depending on whether you wish to use Firebase Hosting or Firebase Functions as a RESTful API.

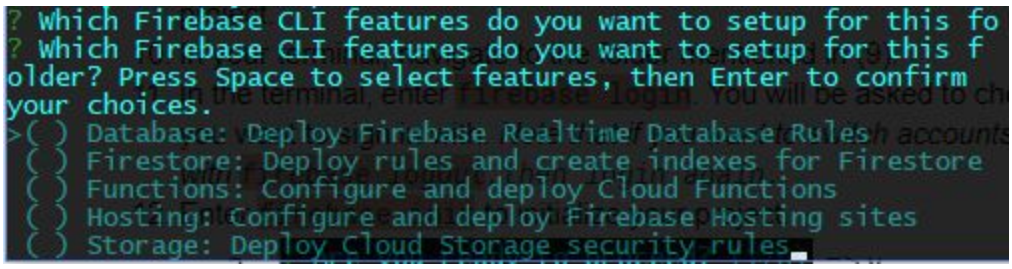
1. Go to https://console.firebase.google.com
2. If you're not logged into a Gmail account, you will be asked to do so
3. If you're signed in to multiple gmail accounts, you can choose which one you want to use for Firebase Hosting. Click on the profile picture (top-right) and select an account.
4. Click on an existing Firebase project that you wish to use. If you don't have any project yet, you can create one by clicking on "Add project". If you're creating a new project:
    a. Write a project name of your choice under "Project name"
    b. Under "Project ID", you can define an ID for your project. It may be useful to use a meaningful ID as Firebase will use this to generate URLs to your data.
    c. Read and select the check boxes and create the project.
5. Once your Firebase project opens, click on the "Develop" tab in the sidebar.





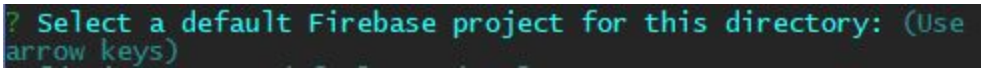

6. Click on “Hosting” or “Functions”.
7. On the next window, press “Get started”. This brings up an instruction dialog that shows you what to do. We’re still going to go over those steps here :)
8. Open your terminal and type in `npm install -g firebase-tools` then press “Enter”. This will install the necessary packages for working with Firebase.
9. Create a folder on your computer, at the location where you want to store your Firebase project.
10. In your terminal, navigate to the folder mentioned in (9).
11. In the same terminal, at the same location, enter `firebase login`. You will be asked to choose a gmail account you want to sign in with. *Note that if you want to switch accounts, you can always logout with* `firebase logout`, *then* `login again`.

## A.1. Firebase Hosting

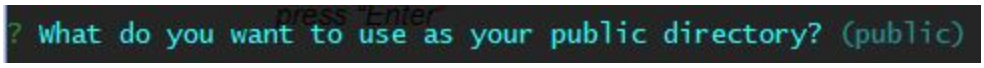
1. Enter `firebase init` to initialize your project.
   a. `? Are you ready to proceed? (Y/n)` => y
   b. 


   => You can navigate with your “up” and “down” arrow keys. Navigate down to “Hosting”, press the Spacebar and press “Enter”. *Note that you can select multiple services for one project. Simply navigate to the service you want, press*

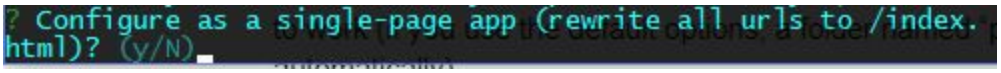
*the Spacebar to select it. Select each service that you need, and when you're done, press "Enter".*

c. 

=> use the arrow keys to navigate to the project that you've created earlier, then press "Enter"

d. 

=> you can stick with the default option here. Press "Enter". This means that your website will need to be under a folder named "public" in order for the deployment to work (if you use the default options, a folder named "public" will be created automatically). If you'd want to use another name, enter it there and press "Enter".

e. 

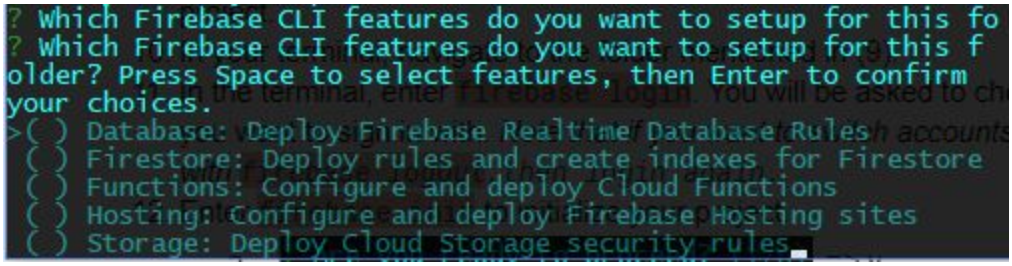
=> press "Enter".

2. Your project is ready for deployment. You can add content under the "public" folder (or whatever you called it). **Make sure that your entry point is named "index.html".** When you deploy to Firebase, the hosting server will look for a file called "index.html" which it will use as homepage for your website. You can add as many html, JavaScript, CSS,... files as you want. Just make sure that there is a file called "index.html" from which everything can be accessed.
3. Once you have all the necessary files in your "public" folder, open your terminal and type in `firebase deploy` to deploy to Firebase.
4. Once the deployment is complete, you will see a "Hosting URL" with a URL where you can access your recently uploaded website. It will usually be ***"https://<your_project_id>.firebase.com"***
5. If you want your website to be accessible through a custom domain name that you own, go back to your Firebase Hosting dashboard (it will appear differently now). Then click the "Connect domain" button and follow the instruction to prove ownership of the domain name. Just note that it may sometimes take a little bit of time for Firebase to be aware of the changes in DNS that you've made on the side of your domain name provider.

## A.2. Firebase Functions for RESTful API

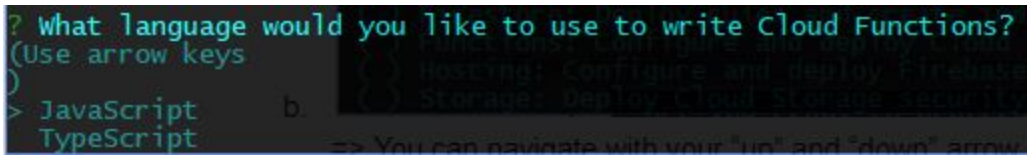
1. Enter `firebase init` to initialize your project.

a. `? Are you ready to proceed? (Y/n)` => y

b. 


=> You can navigate with your “up” and “down” arrow keys. Navigate down to “Functions”, press the Spacebar and press “Enter”. *Note that you can select multiple services for one project. Simply navigate to the service you want, press the Spacebar to select it. Select each service that you need, and when you’re done, press “Enter”.*

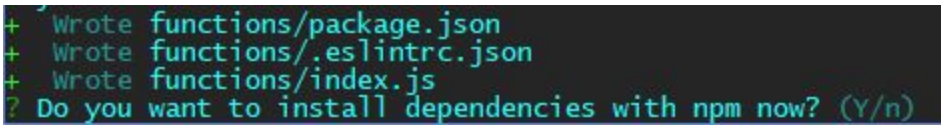
c. `? Select a default Firebase project for this directory: (Use arrow keys)` => use the arrow keys to navigate to the project that you’ve created earlier, then press “Enter”

d. 


=> select the language you’d like to write your functions in. TypeScript is very useful because of the type checking that it introduces. But good ol’ JavaScript is sufficient.

e. `? Do you want to use ESLint to catch probable bugs and enforce style? (y/N)`

=> ESLint will help you catch potential errors in your code before it allows you to deploy it. So you may want to go with “y”.

f. 

=> y

g. If you check your project folder, you’ll see that a “functions” folder has been created. Within that folder, the “index.js” file is the entry point to your functions.

2. Create functions to export in the “index.js” file:
   a. To tell Firebase that you want to export a certain function, you need to define it like this. `exports.nameOfAFunctionToExport = /* function goes here */`

b. To define a REST function, we'll need to use the "firebase-functions" library. Make sure it is imported in your project like this

```
const functions = require('firebase-functions');
```

or if you're using ES6, like this

```
import functions from 'firebase-functions';
```

.

c. You need to use the https request, like this

```
exports.giveANameToThisFunction = functions.https.onRequest((request, response) => {
    //function body
});
```

You can name "request" and "response" whatever you'd like. Just remember that the first argument is going to contain the user's request, and the second will be used to send a response to the user.

d. To get any parameter passed through the URL (like the string "Nick" or the number 24 in this REST request: *www.website.com/nameOfAService?**firstName=Nick&age=24***), you can use "request.query.<name_of_the_parameter>". For instance, for the query above, you could get the string "Nick" by doing "request.query.firstName" and the number 24 by doing "request.query.age, like so:

```
exports.nameOfAService = functions.https.onRequest((request, response) => {
    const fName = request.query.firstName;
    const age = request.query.age;
});
```

e. To send a response to the user, you can use the "response" parameter to send a JSON object back to the user, like this:

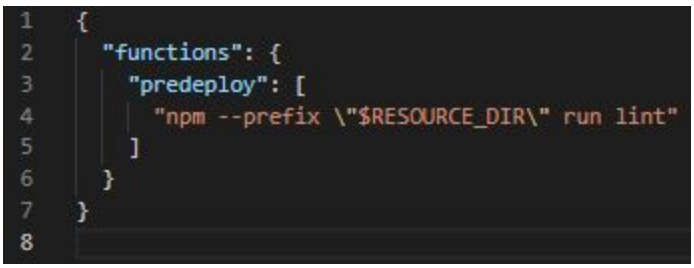
```
exports.getUsername = functions.https.onRequest((request, response) => {
    const fName = request.query.firstName;
    const age = request.query.age;

    let username = fName + age;
    response.json({
        result: username
    })
});
```

You can add as many fields to the response object as you'd like.

3. You can add as many functions as needed. Just make sure that for each of them that you want to export you use "exports." followed by the name you want them to have.
4. Once you're done updating your functions, go back to your console and type `firebase deploy` to deploy them.

a. **Important note for Windows Users**: If you're using the lint, you may have some minor issues with the file "firebase.json". It will be generated with a syntax that may cause some issues when trying to deploy functions from the Windows Shell Command. To fix it, navigate to the file called "firebase.json" from the root of your project folder. The content of the file should look something like this



```
{
  "functions": {
    "predeploy": [
      "npm --prefix \"$RESOURCE_DIR\" run lint"
    ]
  }
}
```

. You need to change this `\"$RESOURCE_DIR\"` into this `\"%RESOURCE_DIR%\"`, so that your Windows command line will be able to understand that it's referring to a variable.

5. If the deployment is successful, you will get a link in the command line showing you the URL by which to access your functions. You can also see those by going back to your Firebase Functions dashboard. For each [http] deployed function, you will see a link corresponding to the URL by which to invoke those functions.
6. If you edit your index.js and redeploy, any functions you will have removed from the file will be deleted from the server, and any functions you have added on the file will be added on the server.