# 5. Modifications of the Original

## The Original Plan

The system was out to see a static schema and map that would allow the user to move from one point to another within the university. The point here was to have an administrator that would add maps to the systems storage. Most of the original work was image based and would see a physical relation between the creator of a map and the physical location. The knowledge base for such a system was considered of top notch importance as the coding end of the application called for knowledge and imagery recognition of this environment.

The system was out to have the location of the user mapped directly to the map images as coded and hence provide the location and path finding facility as aimed by the map.

All this, as workable as it was deemed, would require great lengths of fine tuning of the system and map over time. While the original development plan majored on the testing of the system for both its efficiency and effectiveness, the plan thus took on a different course that would have it major on the testing. One that had the map manually provided by the system and not the user or/and developer.

Such a case called for dynamic and at-runtime map creation. One that proved effective to use even outside the university.

This saw research that led to different findings, among them

* Adding polylines to google maps (Jaquez, 2020)
* Adding markers to google maps (Jaquez, 2020)
* Creating route calculators in google maps (Biswas, 2020)
* Dynamic Polylines embedded on google maps, Flutter (Agutu, 2020)
* Flutter – Google maps, makers and polygons (Raja Yogan, 2018)

## The New Plan

The new course saw the development focus on dynamic map creation. The idea was to “mount” customized maps on top of already existing maps, say google maps. The approach here took into account the fact that google maps or any other provider at that, has fixated maps that only they can modify. These maps lack a modification package that would allow a user to add routes and points to what would go out as very remote and/or private areas. These includes mini-shops, homesteads, junctions, what have you. The direction taken to have such an application would allow a user to add a marker/point, and create a route as they navigate with any means of transportation; here, the route they take is drawn via polylines and the system allows them to add more points along the way.

This approach would however need to restrict only the administrators onto adding maps, while evidently allowing other users to share their locations with each other while at it.

This effort saw the making of the following points as the strongholds and major milestones of creating this system, and any other of its kind.

### Authorization - functionality

* Login - UI and Functionality
* Sign up – UI and Functionality
* Forget Password – UI and Functionality
* Navigation Drawer – UI and Functionality
* Manage Users

### Mapping and Pathfinding

* Map View
* Map View State - User/Admin
* User -> Search and Share location
* Admin – Share location and add Maps
* Bound Location (Lifted due to testing... didn't make sense to bound the app to the SUCC location while the idea can work beyond the university.

## Requirements

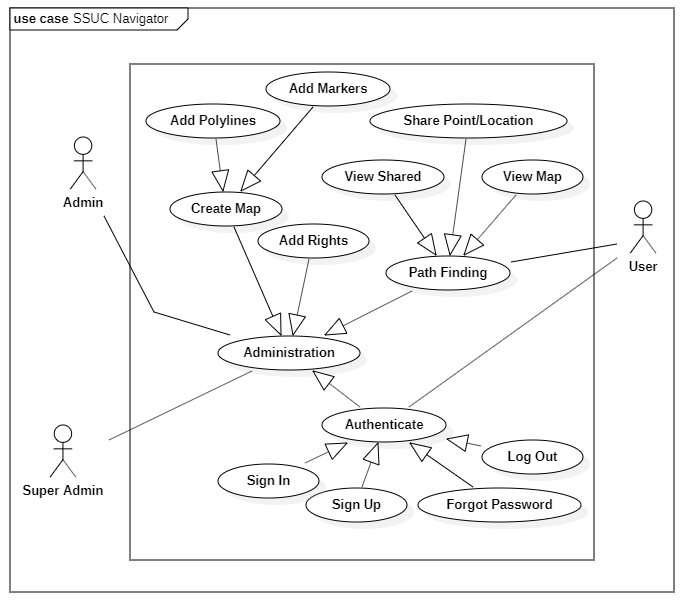
The requirements for the development was centered on Flutter, an open-source UI software development kit created by Google. Used to develop applications for Android, iOS, Linux, Mac, Windows, Google Fuchsia, and the web from a single codebase. The development was however based on android system, in as much as the development platform could evidently span out to more.

The system and its development required these major components;

* Polylines
* Markers
* Directions
* Google maps
* Authentication
* Offline first Storage

## Design

The following is a use case diagram giving an imagery description of the systems design.



The app is initiated by a super admin who then adds other admins, thus giving them the administrative rights as outlined in the above use case. The above minimized the systems development to only but one login system. This is as, the different user types were given different authoritative rights to in app processes, the application was based to thus authorize user functioning, rather than authenticate user identity (which it still was able to achieve through one login system).

The next chapter goes in deeper detail on the implementation of the above use case for the system in regard.

# 6. Implementation Framework and Details

## The Storage

The storage as was used in development, was geared to factor in two major features.

1. *Offline-first*

In offline-first approach, local storage data is used as the main data source, and the data is continuously synchronized with a remote database. If an interruption occurs, the synchronization and database reboot itself once the connectivity is back.

This approach proved best to be put in use as a way to provide the end user with data even while offline. This data includes but is not limited to; points shared to them, and maps (polylines and markers) added by the administrators.

1. *Non-SQL*

The development majored on the dynamic nature of non-SQL databases. Data is stored in many ways. They can be column-oriented, document-oriented, and graph-based or organized as a Key-Value store. This flexibility means that:

* One can create documents without having to first define their structure
* Each document can have its own unique structure
* The syntax can vary from database to database
* You can add fields as you go

The storage was thus provided with Firebase’s Cloud Fire store that allows one to store, sync, and query data for your mobile and web apps - at global scale.

The database structure was as follows,

### Authentication

The authentication was provided by firebase authentication, with the defaults for both,

* Login with Email and Password
* Login with Gmail

The defaults were as follows;

1. Identifier of type email
2. Provider
3. Created of type Date
4. Signed\_in of type DateTime
5. UserUID of type Varchar
6. UserPassword of type Varchar

### Shared

This set was made up of points shared by users to other users, and was structured to take;

1. Address of type Varchar
2. Markers of type list/array
   1. Latitude of type int
   2. Longitude of type int
   3. Altitude of type int
3. Receiver of type email
4. Sender of type email

### Users

This set was made up of authenticated users, geared to providing authoritation and was structured to take;

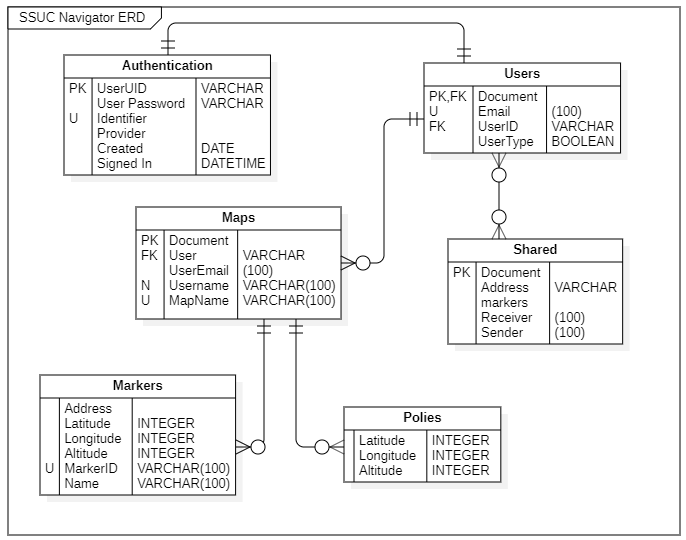
1. Document with reference to UserUID
2. Email with reference to User Email
3. UserType of type bool; either a **guest** or an **admin**

### Maps

This set was made up of maps created by the administrators, and was structured to take;

1. Document, Unique identifier to the map
2. Map of type list/array
   1. Markers of type list/array
      1. Latitude of type int
      2. Longitude of type int
      3. Altitude of type int
3. Polies of type list/array
   * 1. Latitude of type int
     2. Longitude of type int
     3. Altitude of type int
4. User of type Varchar with reference to Authenticated user
5. UserName of type Varchar and is null-able
6. UserEmail of type email
7. MapName of type Varchar

The below ER diagram shows the relation between the above aid out storage components. Notice the user Identifier from the authentication component throughout all other components where applicable and necessitated.



## The User States

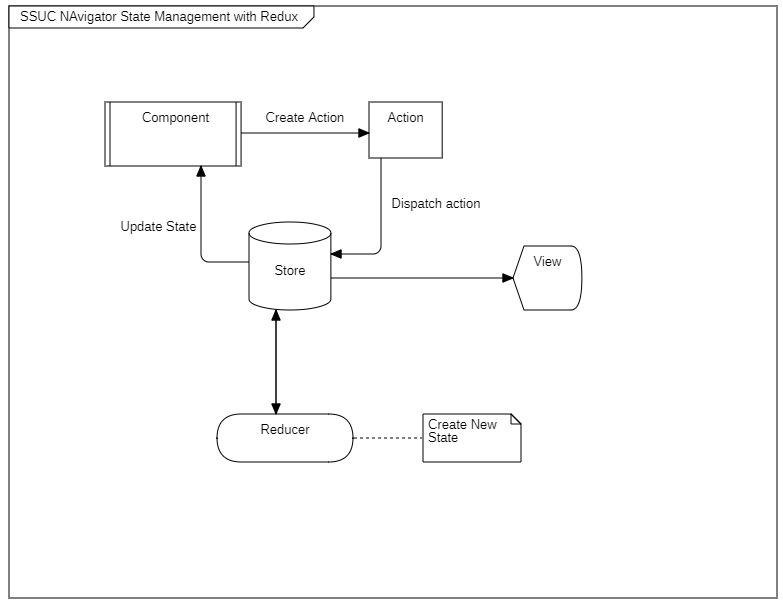
The state as was managed throughout the app saw the implementation and employment of the Redux state manager in Flutter.

With redux, and as was implemented, an action is dispatched to the store. Initially, this action is to find the user type and either display or hide certain elements all with regard, great regard, to the user type.

The store in sync with the reducers, creates a new state with respect to the available and present data.

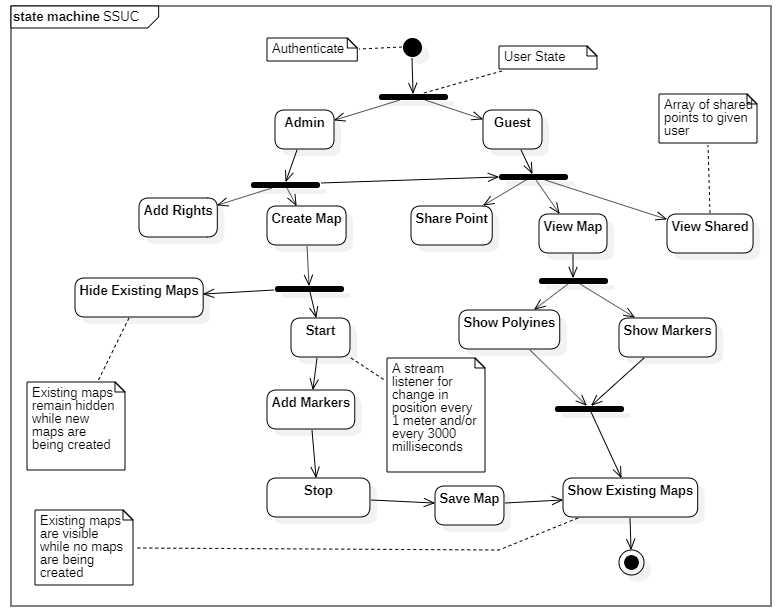
This new state is pushed to the component that originally initiated the action and thus displayed to the front end.

The below flow gives a descriptive image of this sequence.



Taking reference from the use case diagram from the previous chapter, the map-view as an example, employs redux to manage its state. Here, if the user is an admin, they are provided with an end-point that allows them to create maps. This very end-point is in itself nested with other states all managed by redux. If the user is of type **guest**, redux provides the user with a view that is void of this end-point.

The state for the whole system is recorded from the bellow state diagram.



## Coding Decisions

The code decisions made had the following major components at their core;

### Current location

The current location gets the device’s current geographical point. This necessitates the app to access location privileges from the user of the device. This location is fed to the view that displays the map as will be shown later in this section.

This also takes with it the zoom property for the view of the map.



### Address

The bellow function is used to generate an address for the location from the values of this very location. For the system, the development used the **name, locality** and **postal Code.**

Other available options include,

* place.administrativeArea
* place.country
* place.isoCountryCode
* place.position
* place.subAdministrativeArea
* place.subLocality
* place.thoroughfare
* place.subThoroughfare



### Polylines

This is the provided feature of any polyline. With it comes,

1. Poyine ID
2. Color of the polyline
3. An array of the polyline points
4. The width of the polyline



### Start Marking

### Mark Point

This adds a marker to the current position of the user.



### Get Position Stream

The creation of polylines gets an array of position components. This array is provided or rather created by a stream that listens to the changes in the position of the device based on two conditions,

1. The distance, where the stream adds a new point to the array if there is a change in position every one meter.
2. The time, where the stream adds a new point to the array if there is a change in position every 3000 milliseconds.

This is all subscribed to a ***positionSubscription*** that is later on used by the stop action to change the state and stop this very stream once the user (administrator) is ready to save this map instance.



### Stop Marking

### Stop Stream

This is a ca to stop streaming the user’s location. After it is called, the app pops to its previous view and hides the end-point for map creation.



### Save Map

The map as created by the administrator is saved with regard to the earer touched schema of the storage, with a **mapname,** a set of **polylines,** a set of **markers,** the user identifier – **userUID,** the **username,** and the user’s **email.**



### The View



Notice two components of this view, the markers and polylines. The two heavily depend on the state as held by redux as discussed earlier. In the state that has the administrator creating maps for example, sets the polylines from within the material app (**mypolylines.values)** , else, when no map is being created, the polylines from the store (**store.state.polylines**) are the ones in display.

## Tools and Components

As mentioned before, the system and its development thus, was based on Flutter. Beyond that, the following was the environment over which this development fully depended on.

**Operating System**: Microsoft Windows 10

**Build**: Android OS

**Frame Work and library**: Flutter

**Language**: dart

The packages and plugins used were, but not limited to:

* async
* boolean\_selector
* characters
* charcode
* clock
* cloud\_firestore
* cloud\_firestore
* collection
* convert
* crypto
* equatable
* fake\_async
* firebase\_auth
* firebase\_core
* firebase\_storage
* flare\_dart:file
* flare\_flutter
* flutter
* flutter\_polyline\_points
* flutter\_redux:file
* flutter\_test
* geolocator
* google\_maps\_flutter
* google\_sign\_in
* http
* http\_parser
* intl
* js
* location
* matcher
* meta
* path
* pedantic
* quiver
* redux
* sky\_engine
* stack\_trace
* stream\_channel
* string\_scanner
* term\_glyph
* test\_api
* toast
* typed\_data
* vector\_math