GSoC 2021 Proposal

Monumento

Suryansh Singh Tomar

Table of contents

Personal Information

Name: Suryansh Singh Tomar
University: Indian Institute of Technology (BHU), Varanasi
Field of Study: Mining Engineering
Date of Enrollment: July 2019
Expected Graduation date: July 2023
Degree: Bachelor of Technology
Year: Sophomore
Github: https://github.com/PaRaDoX50
LinkedIn: Link
Gitter nick: PaRaDoX50
Email Address: suryansh.stomar.min19@itbhu.ac.in
suryanshtomar.st10@gmail.com
Phone number: (+91) 7000037559
Timezone: Indian Standard Time (UTC +5:30)

Commitments

- **How many hours will you work per week on your GSoC project?**

  I am planning to spend 40 - 50 hours or more on the project per week.

- **Do you have access to Mac and iPhone for iOS development?**

  Yes.

- **Other Commitments**

  I have no other commitments during the GSoC period.

- **Do you plan to apply for any other organisation for GSoC'21?**

  I am only applying to Aossie for GSoC'21 and have no plans to contribute to any other organisation.

- **If you're selected as a GSoC student, would you like to work on other tasks besides the projects of your choice?**

  Yes, I would love to work on other tasks that are not related to my GSoC project.

- **If you're not selected as a GSoC student, would you like to work on the projects as a general contributor?**

Yes, even if I'm not selected as a GSoC participant, I would happily continue working as a general contributor.

- **Would you like to contribute to Aossie in the long term, after the GSoC program ends?**

  Yes, I would like to contribute to Aossie even after GSoC ends.

- **What motivated you the most towards applying for GSoC?**

  There are various reasons for which I wanted to apply for GSoC but my main motive was to get recognised as a GSoC participant. Also, the stipend was not a motivating factor but an opportunity to work with a big organisation like Aossie was.

Introduction to the Project

Monumento is an app that lets you detect monuments and visualise their 3D models on your screen excellently and interestingly using AR.

Technologies used by the app at present

- Monumento is a Native Android app with most of the screens built using Flutter.
- A Flutter module is embedded in an Android App. Flutter and Native Android shake hands through the Method Channels provided by Flutter.
- Native Screens are used for core features - Detecting Monuments using CloudVisionApi, Rendering Models of Monuments using SceneForm SDK.
- All other screens like Authentication, Profile, Home etc. are made using Flutter.
- Firebase is used for authentication, and Firestore for the database.

Scopes of Improvement

- Since most of the app is built using Flutter, we should make full use of it and make the app support iOS devices as well.
- The app can be of more use if we add a section that will act as a Social Media Platform for Travellers.

- The Business-Logic/API-calls and the UI part of the app need to be separated correctly. Right now, All the API calls are made from the UI class itself.
- The UI/UX of the app also need improvements, especially the Profile and Intro screens. And the app also lacks some basic features like Form validation, Profile Picture selection during the Registration process, Using Gallery as a Source for image at the time of Monument Detection etc.

My Primary Goals

- Adding support for iOS devices. (Sub Project 1)
- Adding a section that will act as a Social Media Platform for Travellers. (Sub Project 2)
- Refactoring the code and restructuring the app into a suitable Pattern/Architecture (Preferably BLoC). (Sub Project 3)
- Improving the overall user experience by implementing some basic features and fixing all possible bugs as stated in the Scopes of Improvement Section. (Sub Project 4)

I plan to implement my primary goals in order Sub Project 3 -> Sub Project 1 -> Sub Project 2 because implementing Part 3 at first will set up an architecture for the whole app to follow.
I plan to implement Part 4, i.e. fixing bugs and adding the basic features, during the community bonding period and in between the implementation of other parts.

Why Monumento?

The day I installed Monumento on my device, I kept using it because I enjoyed it. Roaming around the house to find a horizontal plane and then rendering a massive Monument was altogether a fun experience.
I always wanted to contribute to a big organisation like Aossie and to a project I liked using.
Moreover, I love working with Flutter and Native Android. Therefore Monumento was my go-to project.

Implementation of the Project Goals

Adding support for iOS devices

Plan of Action

Technologies I plan to use

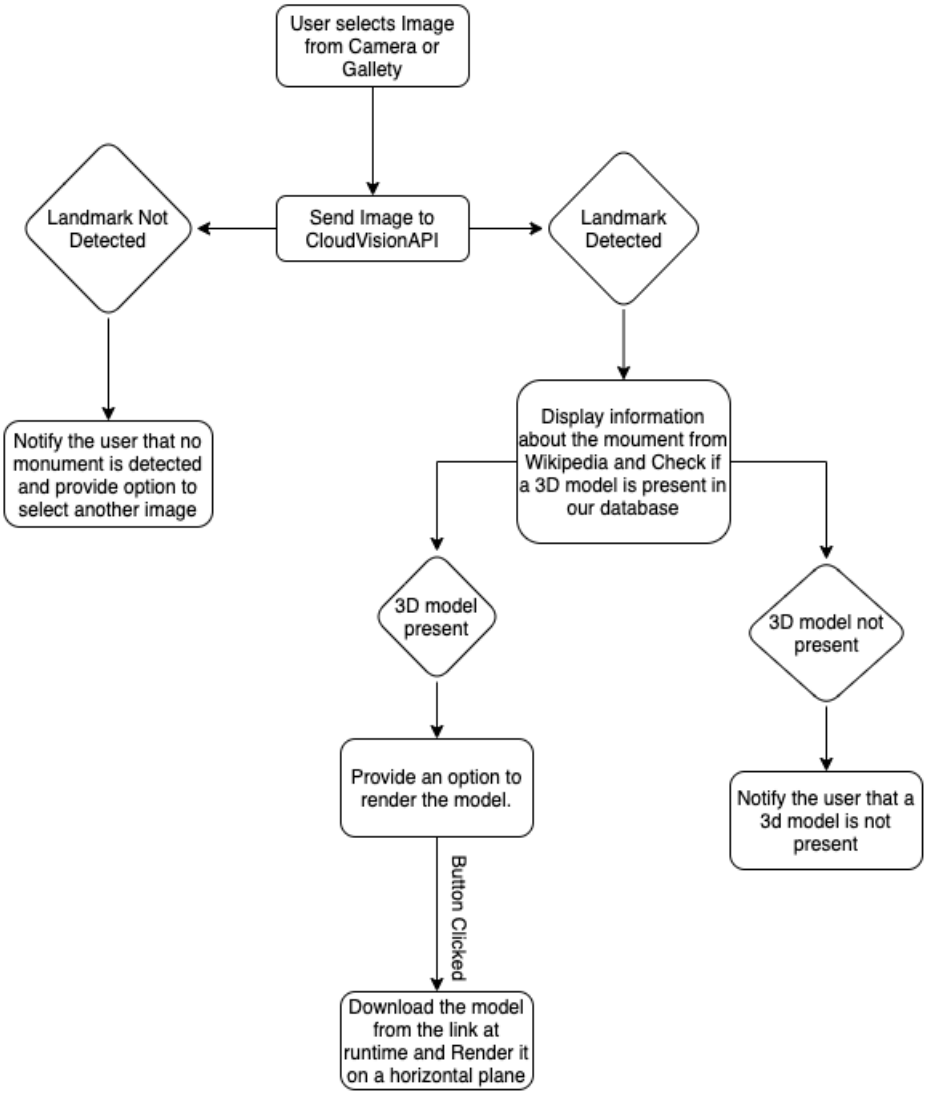
1. **Swift** with **UIKit App Delegate** (Flutter Module supports UIKit App Delegate and there is no documentation for other alternatives like SwiftUI) (Click here for the Flutter + UIKit documentation)
2. **ARKit** with **SceneKit** for AR
3. **Cloud Vision API** for Landmark Detection

The app flow and UI for iOS's core features will be the same as our already built Android app.
The user will be able to detect monuments in an image with the input source being the device's camera or gallery and will be able to render them onto his/her screen. Also, the user will be able to render the already listed popular monuments directly.

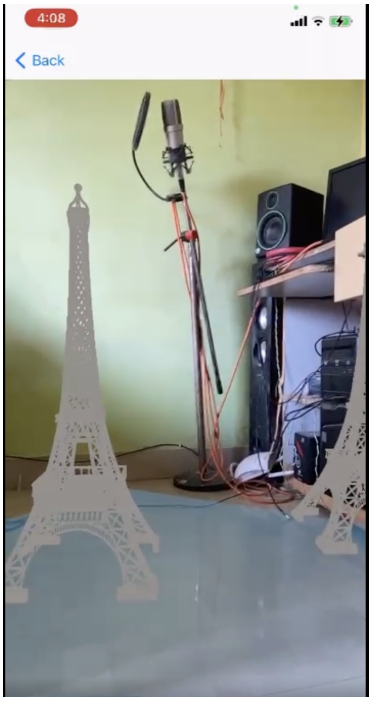
The first step of the implementation will be to embed the flutter module into a new iOS app.
For embedding the flutter module inside an iOS app, I have referred to the official documentation.
There are many ways to embed the module into an iOS app, but the most suitable one is using the **CocoaPods** dependency manager. It allows rapid integration of the up-to-date module inside an iOS app.

Now, before starting to implement the native screen, we will set the **FlutterViewController** as the rootViewController. Setting FlutterViewController as the rootViewController means that the first screen displayed to the user when he/she opens the iOS app will be the flutter screens.
This can be done in the SceneDelegate file.
After the app has started, as soon as the user clicks the button to detect monuments, he/she will be taken to the native screens. The navigation from the flutter screens to the native screen will be achieved by setting up the method channels.

This will be the overall flow for the Monument Detection (Native Screen) part



The users will be able to take a picture or select one from the gallery.
For the detection of the landmarks/monuments, we will use CloudVisionApi, which is just another REST API that uses HTTP POST operations. For the networking operations, I plan to use Alamofire, a popular 3rd party networking library for iOS.
Once a landmark is detected, we will get the link for its model from our pre saved static data and download the model at runtime. Once the model is downloaded, we will use **ARKit** and **SceneKit** to render it on the screen. For learning about rendering 3d models on a horizontal plane, I referred to this ARKit Tutorial.



Rendered 3D model will look something like this.

I have already implemented a major part of this section and also made a W.I.P MR iOS Implementation.

Reference Links

https://flutter.dev/docs/development/add-to-app/ios/project-setup
https://flutter.dev/docs/development/add-to-app/ios/add-flutter-screen?tab=engine-swift-tab
https://www.appcoda.com/arkit-horizontal-plane/
https://github.com/Alamofire/Alamofire

Detailed Implementation

The implementation will include three parts.

1. Embedding the Monumento Flutter Module to a new iOS app, setting up the FlutterEngine, FlutterViewController and Method Channels for the iOS app
2. Detecting Monuments using Cloud Vision API
3. Augmenting 3D models on a horizontal plane using ARKit and SceneKit for iOS

Adding the Monumento Flutter Module to a new iOS app, connecting the FlutterEngine, FlutterViewController and Method Channels with the iOS app

Embedding the Monumento Flutter Module to a new iOS app

Flutter module can be implemented using the CocoaPods dependency manager and installed Flutter SDK
First, we will have to create a new iOS app and place it in the project directory.
Then, we will have to create a PodFile and Specify the path for the monumento_module.
Our PodFile will look something like this:

```
platform :ios, '11.0'
flutter_application_path = '../monumento_module'
load File.join(flutter_application_path, '.ios', 'Flutter', 'podhelper.rb')

target 'ios-monumento' do
  use_frameworks!
  install_all_flutter_pods(flutter_application_path)
end
```

After running pod install in the iOS app directory, our module will be ready to work with the iOS app.

Setting up FlutterEngine, FlutterViewController(as default ViewController) and Method Channels

According to the official Flutter Documentation - "The FlutterEngine serves as a host to the Dart VM and your Flutter runtime, and the FlutterViewController attaches to a FlutterEngine to pass UIKit input events into Flutter and to display frames rendered by the FlutterEngine."

The piece of code below will start the FlutterEngine and set up the MethodChannels.

```
var flutterEngine = FlutterEngine(name: "my flutter engine")

override func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions:
            [UIApplication.LaunchOptionsKey: Any]?) -> Bool {
    // Runs the default Dart entrypoint with a default Flutter route.
    // 1
    flutterEngine.run();
    // Used to connect plugins (only if you have plugins with iOS platform code).
    GeneratedPluginRegistrant.register(with: self.flutterEngine);

    // 2
    let monumentDetector = FlutterMethodChannel(name: "monument_detector",
                                                binaryMessenger: flutterEngine.binaryMessenger)

    // 3
    monumentDetector.setMethodCallHandler({
        [weak self] (call: FlutterMethodCall, result: FlutterResult) -> Void in
        // Note: this method is invoked on the UI thread.
        guard call.method == "navMonumentDetector" else {
            result(FlutterMethodNotImplemented)
            return
        }

        // 4
        let storyboard = UIStoryboard(name: "Main", bundle: nil)
        let vc = storyboard.instantiateViewController(withIdentifier: "MonumentCaptureViewController")

        self?.flutterEngine.viewController?.navigationController?.pushViewController(vc, animated: true)

    })
    return super.application(application, didFinishLaunchingWithOptions: launchOptions);
}
```

Walking through the code line by line:

1. flutterEngine.run() starts the FlutterEngine.
2. MethodChannel is created with the name "monument_detector" which will be used to navigate to the native screens of iOS from Flutter Screens.
3. We set up a method call listener, which will listen to the method calls from Flutter code.
4. If the method navMonumentDetector is called from the Dart code, then the app will navigate to the native MonumentCaptureViewController just like we navigate to MonumentDetector Activity in Android.

Now, We will also have to set FlutterViewController (Flutter Screens) as the launching or root screens for the iOS app. We will have to change the rootViewController to FlutterViewController in the SceneDelegate class.
The piece of code below will do the job.

```
var window: UIWindow?

func scene(_ scene: UIScene, willConnectTo session: UISceneSession, options connectionOptions:
                    UIScene.ConnectionOptions) {

    guard let windowScene = scene as? UIWindowScene else { return }
    // 1
    let flutterEngine = (UIApplication.shared.delegate as! AppDelegate).flutterEngine
    // 2
    let flutterViewController =
        FlutterViewController(engine: flutterEngine, nibName: nil, bundle: nil)
    // 3
    let navVC = UINavigationController(rootViewController: flutterViewController)
    window = UIWindow(windowScene: windowScene)
    // 4
    window?.rootViewController = navVC
    window?.makeKeyAndVisible()
}
```

Walking through the code line by line

1. Accessing the flutterEngine defined in the AppDelegate file.
2. Creating a FlutterViewController using the flutterEngine.
3. Creating a NavigationController using the previously created FlutterViewController.
4. Setting the previously created NavigationController to the rootViewController of the app.

Detecting Monuments using Cloud Vision API

We will have to capture or select the image for detection. It can be easily done using the inbuilt UIImagePickerController. Now, before passing the image to CloudVisionAPI, it needs to be converted to Base64 String. After converting the Image to Base64 String, it will be passed to the CloudVisionAPI.
The method (which accepts Base64 Encoded Monuments's Image and a function "completion") will call the Cloud Vision's API(Which is a REST API) with the HTTP POST operation.

```swift
private func callGoogleVisionAPI(
        with base64EncodedImage: String,
        completion: @escaping (DetectedLandmark?) -> Void)  {

        // 1
        let parameters: Parameters = [
            "requests": [
                [
                    "image": [
                        "content": base64EncodedImage
                    ],
                    "features": [
                        [
                            "type": "LANDMARK_DETECTION"
                        ]
                    ]
                ]
            ]
        ]
        let headers: HTTPHeaders = [
            "X-Ios-Bundle-Identifier": Bundle.main.bundleIdentifier ?? "",
        ]
        // 2
        Alamofire.request(
            apiURL,
            method: .post,
            parameters: parameters,
            encoding: JSONEncoding.default,
            headers: headers)
            .responseData { response in
                if response.result.isFailure {
                    completion(nil)
                    return
                }
                guard let data = response.result.value else {
                    completion(nil)
                    return
                }
                let ldResponse = try? JSONDecoder().decode(GoogleCloudLDResponse.self, from: data)
                // 3
                completion(ldResponse?.responses[0].detectedLandmarks[0])

            }

    }
```

Walking through the code line by line.

1. Setting the "LANDMARK_DETECTION" feature and the base64 image as "image" for parameters.
2. Using the Alamofire Http Networking Library for calling the CloudVisionAPI.
3. Calling the "completion" callback function and passing the detected landmark name as an argument after the call to the CloudVisionAPI is completed successfully.

The passed landmark's name will be used to get the link for its 3D Model from the data we have locally stored as a map (same as we do it in android).
The link will then be used to download the model from the internet. The process is explained in the next section.

Displaying 3D Models of Monuments on a Horizontal Plane

Implementation of this part will include these steps (In order):

1. Downloading the 3D Model.
2. Detecting a Horizontal Plane.
3. Rendering the model on to the screen.

Downloading the 3D Model

Currently, we get our models from Google's Poly which is going to be deprecated on 30th April 2021. Therefore, I suggest that we store our models in Firebase Storage or any other best suited alternative.

We will download the 3D Model from the link (using the URLSessionDownloadDelegate) and save it locally using the methods below to reference it later by its file address.

We will start the download task using a method that will look something like:

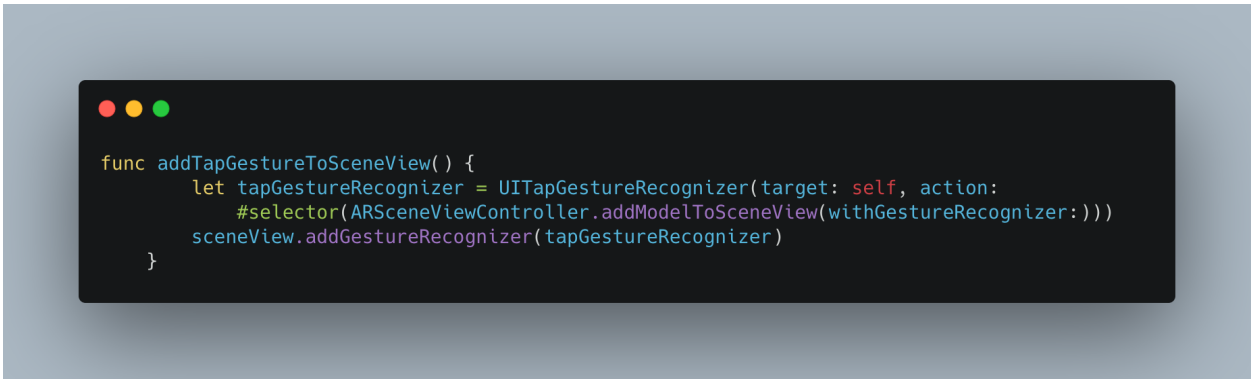
```
func downloadSceneTask(){

    //1. Get The URL Of The SCN File
    guard let url = URL(string: monumentModelURL) else { return }

    //2. Create The Download Session
    let downloadSession = URLSession(configuration: URLSession.shared.configuration, delegate:
            self, delegateQueue: nil)

    //3. Create The Download Task & Run It
    let downloadTask = downloadSession.downloadTask(with: url)
    downloadTask.resume()


}
```

When the download finishes, the method urlSession will be called. It will save the file locally as "model.Extention".
The urlSession method will also add a TapGestureRecognizer to the SceneView after the file is saved locally. The TapGestureRecognizer will handle all the taps on the screen. It will call a method called addModelToSceneView on every tap.
If the user tapped on a detected horizontal plane, the addModelToSceneView method will render the model where the user tapped.

The urlSession method will look something like this below.

```
func urlSession(_ session: URLSession, downloadTask: URLSessionDownloadTask, didFinishDownloadingTo
        location: URL) {

    //1. Create The Filename
    let fileURL = getDocumentsDirectory().appendingPathComponent("model.STL")

    //2. Copy It To The Documents Directory
    do {
        if(FileManager.default.fileExists(atPath: fileURL.path)){
            try! FileManager.default.removeItem(at: fileURL)
            try FileManager.default.copyItem(at: location, to: fileURL)
        }
        else{
            try FileManager.default.copyItem(at: location, to: fileURL)
        }

        print("Successfuly Saved File \(fileURL)")

        //3. Attach a TapGestureRecognizer, so that the model is rendered when user taps a
             horizontal plane
        DispatchQueue.main.async { [self] in
            addTapGestureToSceneView()
        }
    } catch {
        print("Error Saving: \(error)")
    }
}
```

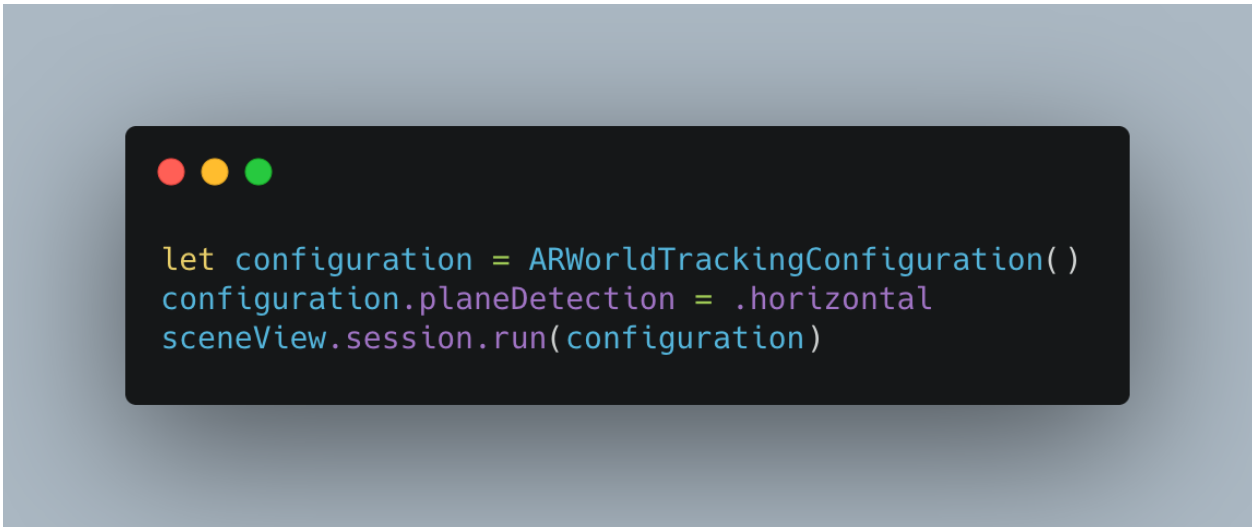
The addTapGestureToSceneView, which will add a TapGestureRecognizer to SceneView will look something like this below:



```
func addTapGestureToSceneView() {
        let tapGestureRecognizer = UITapGestureRecognizer(target: self, action:
            #selector(ARSceneViewController.addModelToSceneView(withGestureRecognizer:)))
        sceneView.addGestureRecognizer(tapGestureRecognizer)
    }
```

The addModelToSceneView is called whenever a Tap is recognized.
The addModelToSceneView method is discussed in more details in the "***Rendering the model on to the screen"*** section.
Detecting a horizontal plane

Now, detecting a horizontal plane in ARKit is simple. We will simply have to add these three lines of code while setting up the SceneView.



```
let configuration = ARWorldTrackingConfiguration()
configuration.planeDetection = .horizontal
sceneView.session.run(configuration)
```

Setting the `planeDetection` property of `ARWorldTrackingConfiguration` to `.horizontal`, this tells ARKit to look for any horizontal plane. Once ARKit detects a horizontal plane, that horizontal plane will be added into `sceneView`'s session.

Every time a new horizontal plane is detected, the renderer method will be called. We will use it to give a visual appearance to the newly added horizontal plane. The implementation of the renderer method will look something like below:

```
func renderer(_ renderer: SCNSceneRenderer, didAdd node: SCNNode, for anchor: ARAnchor) {
    // 1
    guard let planeAnchor = anchor as? ARPlaneAnchor else { return }

    // 2
    let width = CGFloat(planeAnchor.extent.x)
    let height = CGFloat(planeAnchor.extent.z)
    let plane = SCNPlane(width: width, height: height)

    // 3
    plane.materials.first?.diffuse.contents = UIColor.transparentLightBlue

    // 4
    let planeNode = SCNNode(geometry: plane)

    // 5
    let x = CGFloat(planeAnchor.center.x)
    let y = CGFloat(planeAnchor.center.y)
    let z = CGFloat(planeAnchor.center.z)
    planeNode.position = SCNVector3(x,y,z)
    planeNode.eulerAngles.x = -.pi / 2

    // 6
    node.addChildNode(planeNode)
}
```

Walking through the code line by line

1. Here, we create an SCNPlane to visualize the detected horizontal plane (ARPlaneAnchor). A SCNPlane is a rectangular "one-sided" plane geometry. We take the unwrapped ARPlaneAnchor extent's x and z properties and use them to create an SCNPlane.
2. We give colour to the plane.
3. Finally, we add the planeNode as the child node onto the newly added SceneKit node.

Rendering the model on to the screen

Whenever the user taps on a SceneView, the method addModelToSceneView will be called. It will first recognize if the tap is on a horizontal plane or not. If the tap is on a horizontal plane, it will render the model where the user tapped. The method will look something like below.

```
@objc func addModelToSceneView(withGestureRecognizer recognizer: UIGestureRecognizer) {
    let tapLocation = recognizer.location(in: sceneView)
    // 1
    let hitTestResults = sceneView.hitTest(tapLocation, types: .existingPlaneUsingExtent)

    guard let hitTestResult = hitTestResults.first else { return }
    // 2
    let translation = hitTestResult.worldTransform.translation
    let x = translation.x
    let y = translation.y
    let z = translation.z

    // 3
    let fileURL = getDocumentsDirectory().appendingPathComponent("model.STL")
    let modelScene = try! SCNScene(url: fileURL, options: nil)

    // 4
    for node in modelScene.rootNode.childNodes as [SCNNode] {
        node.position = SCNVector3(x,y,z)
        node.scale = SCNVector3(x: 0.005, y: 0.005, z: 0.005)
        sceneView.scene.rootNode.addChildNode(node)
    }
}
```

Walking through the code line by line

1. hitTest method will search for AR anchors(Horizontal Planes in our case) in the captured camera image corresponding to the tapped location in the SceneKit view. If the tap is on a horizontal plane, the code will proceed else it end.
2. We will get the position of the hitTestResult relative to the real world coordinate.
3. Getting the path for the downloaded 3d model and creating a SCNScene using it.
4. Finally, we add the child nodes of the model root node to the root node of the sceneView.

Adding a section in the Flutter Module which will act as a Social Media Platform for Travellers.

Technologies I plan to use

- Cloud Firestore as Database service
- Cloud Functions for updating some meta-information like Item count, last update time for a collection etc
- BLoC architecture pattern because it enhances the scalability and reusability of code.

Features I am planning to implement

- Users will be able to post travel photographs
- There will be a feed where all the posted photographs of the people user follows will be displayed.
- Users will be able to like, comment and share a post.
- Users will be notified whenever someone likes their post or comments on their post.

Database Model for Cloud Firestore

To avoid complex and nested queries, a well-structured database model should be used. Also, without a good database model, we can easily surpass the free usage limit because of many unnecessary queries.

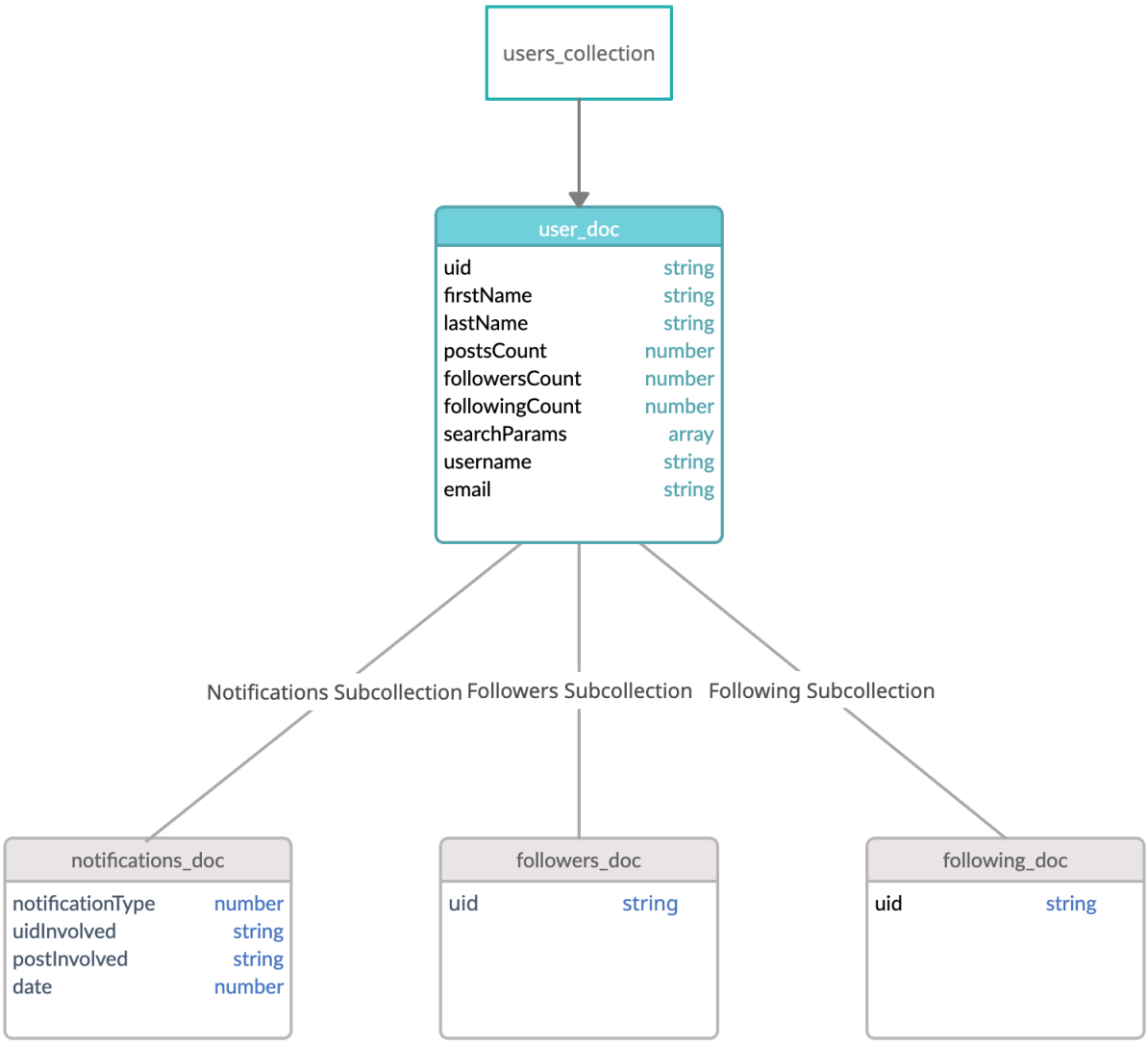
The structure would look something like this :

```
users
   |------- followers      (SubCollection)
   |------- following      (SubCollection
   |------- notifications (SubCollection)

posts

   |-------comments    (SubCollection)
```

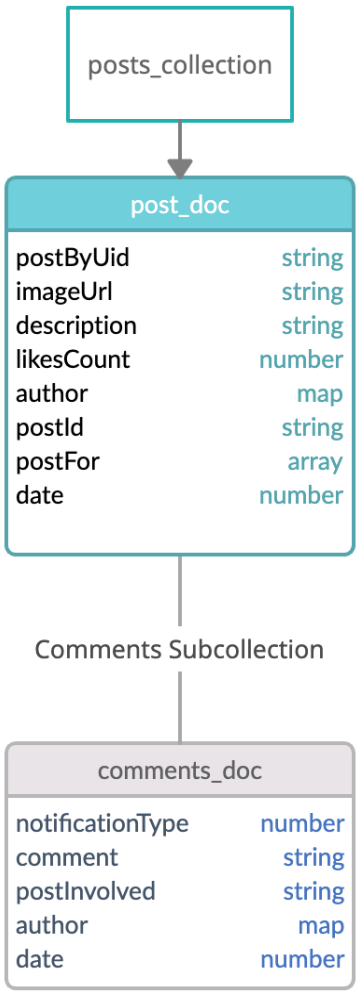
It is better to use SubCollections instead of Master/Root Collections because properties like comments, followers, notifications etc., have a direct connection to their respective document.

We will be able to retrieve all the comments for a post with just a CollectionReference, if we use SubCollections whereas we would have to query results if we use Master/Root Collections.

The "users" collection will look something like this below



The "posts" collection will look something like this below:



Saving duplicate data about the author inside a comment or a post saves us from performing extra queries to get the author's data like profile picture, username etc.

The time taken by firestore to run a query is directly proportional to the number of results/documents it extracts from the query and is independent of the number of documents the collection has. For example, if we run a query that extracts 10 documents, it will take the same amount of time whether we have 6 million or 60 documents in the collection. This is why I plan to keep a "posts" master collection instead of a feed subcollection for every user document.

Plan of Action

There will be 6 screens for the Social Media section.

1. Feed
2. Notification
3. Discover
4. Profile
5. Comments
6. New Post

At present, these are the screens that are placed at the BottomNavBar of the app.

1. Home
2. Explore Monuments
3. Bookmarked Monuments
4. Profile

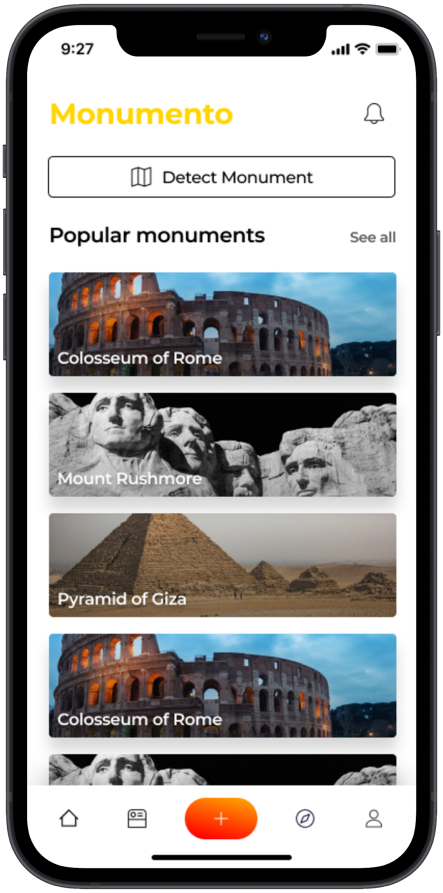
I plan to remove the Explore Monuments and Bookmarked Monuments Screen from the BottomNavigationBar.
After these screens are removed, the user will be able to reach the Bookmarked Monuments Screen by going to his/her profile tab and clicking on the Bookmarked Monuments button.
And the user is already able to explore the popular monuments on the Home Screen.

At the BottomNavigationBar, I plan to keep

1. Home
2. Feed
3. New Post
4. Discover
5. Profile

Proposed designs for these screens can be downloaded from here.

The new design for Home Screen :

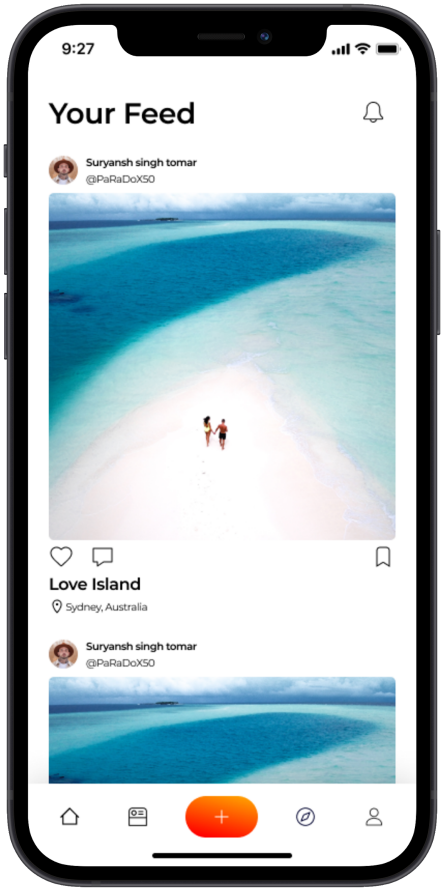


The "Detect Monument" button will take us to the prebuilt native screens.

The detailed implementation of each screen of the social media part is explained in the next section.

Detailed Implementation of the Screens

Feed



Considering Alex to be a user.

In this section, Alex will be displayed

- all the posts from the people he follows
- posts from the people he does not follow but these will displayed after the post from the users he follows are done displaying.

This is the same feed-type Instagram uses.

Every time a user will upload a post, an array field named postFor will also be added to the post document. The field will contain all the user-ids of his/her followers.

We will now use this field to get all the posts of the users that Alex follows.

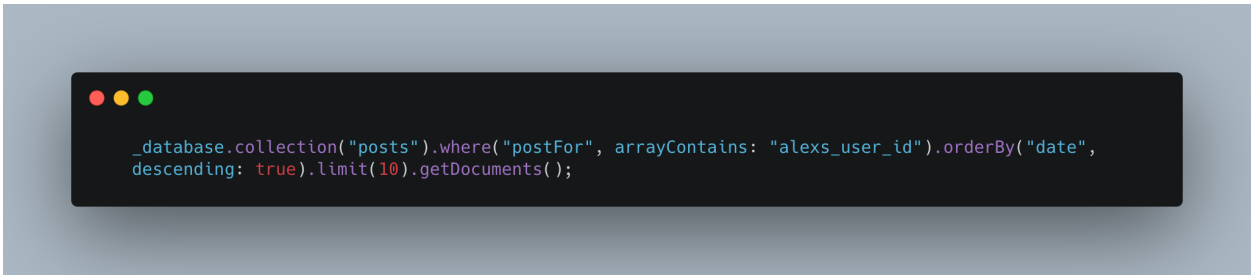
Technically, we will want to retrieve all the post documents which contain Alex's user-id in their postFor array field. This can be achieved using a single simple query.

```
_database.collection("posts").where("postFor", arrayContains: "alexs_user_id").orderBy("date",
descending: true).getDocuments();
```

But this query will at once retrieve all the posts uploaded by the users Alex follows which will be very costly in terms of loading time and data.

We will have to limit the query results and implement lazy loading for the Feed to resolve this problem.

For limiting the results, we can use the .limit() method provided by Cloud Firestore.



```
_database.collection("posts").where("postFor", arrayContains: "alexs_user_id").orderBy("date",
descending: true).limit(10).getDocuments();
```

This query will fetch us 10 documents.

And once the user has scrolled through these 10 posts, we will have to load more posts to display.

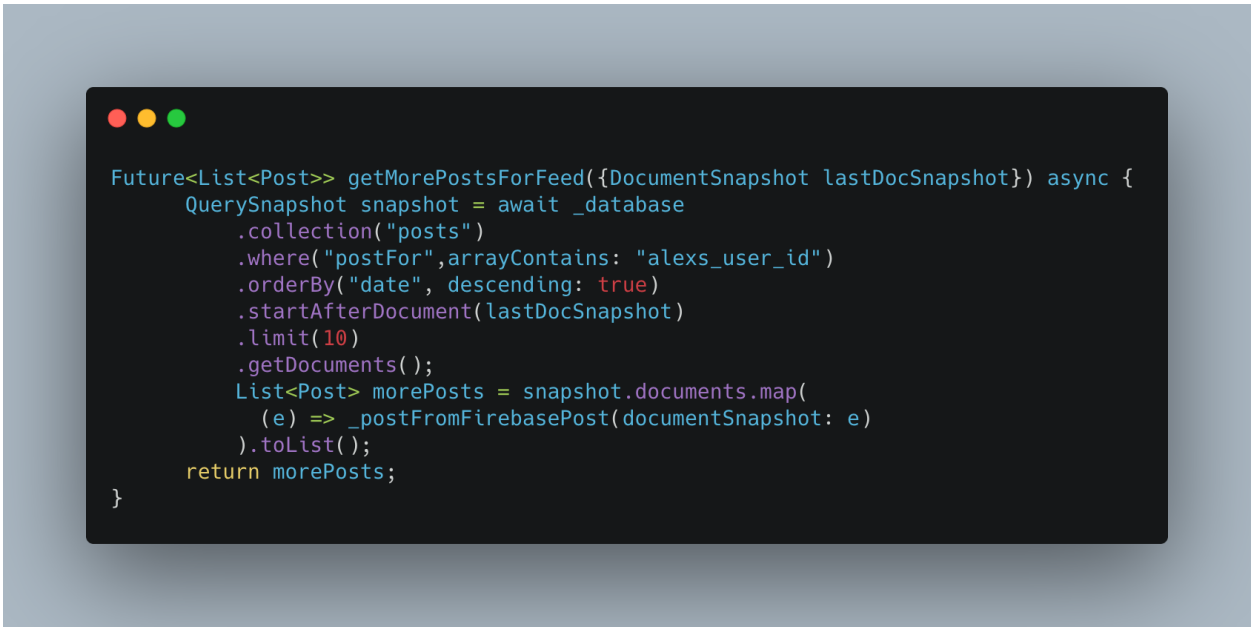
We can achieve this behaviour by either using the lazy_loading plugin or by manually attaching a ScrollController to the ListView and triggering the method to load more posts when the scroll reaches a certain point.

I prefer the plugin because it eliminates all the boilerplate code.

The lazy loading widget would look something like this:

```
LazyLoadScrollView(
    onEndOfPage: () => getMorePostsForFeed(lastDocSnapshot: data.last.doc),
    child: ListView.builder(
      itemCount: data.length,
      itemBuilder: (context, index) {
        return PostTile(data[index]);
      },
    ),
  ),
```

As soon as the user reaches the end, getMorePostsForFeed method will be called. It will look something like the method below:



```
Future<List<Post>> getMorePostsForFeed({DocumentSnapshot lastDocSnapshot}) async {
  QuerySnapshot snapshot = await _database
      .collection("posts")
      .where("postFor",arrayContains: "alexs_user_id")
      .orderBy("date", descending: true)
      .startAfterDocument(lastDocSnapshot)
      .limit(10)
      .getDocuments();
      List<Post> morePosts = snapshot.documents.map(
        (e) => _postFromFirebasePost(documentSnapshot: e)
      ).toList();
  return morePosts;
}
```

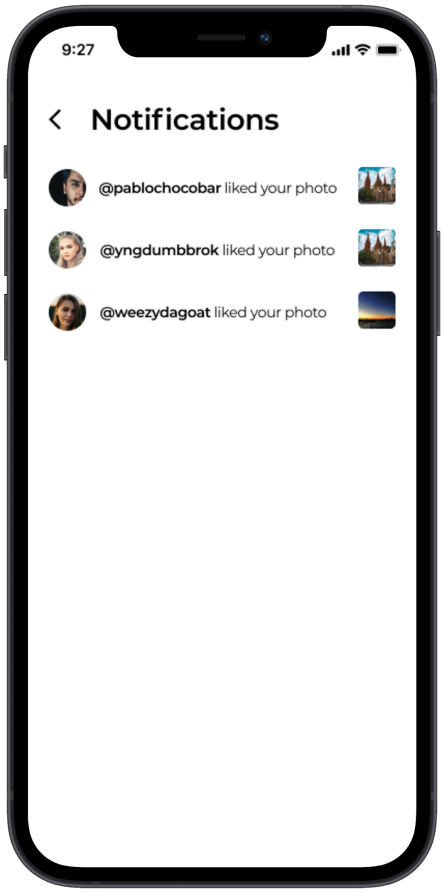
Here the .startAfterDocument method excludes all the documents before the passed lastDocSnapshot. It also excludes the passed lastDocSnapshot.

The methods .limit() and .startAfterDocument() are the reason why pagination/lazy-loading using firestore is easy.

Liking a Post

Whenever a user will click on the "like" button of the post, the user's id will be added in the "likedBy" array field of the post document. Also, the likesCount field will be incremented by 1.

Notifications Screen



This screen will display all the notifications. The notifications for every user will be retrieved from the "notifications" SubCollection in the "user" document.

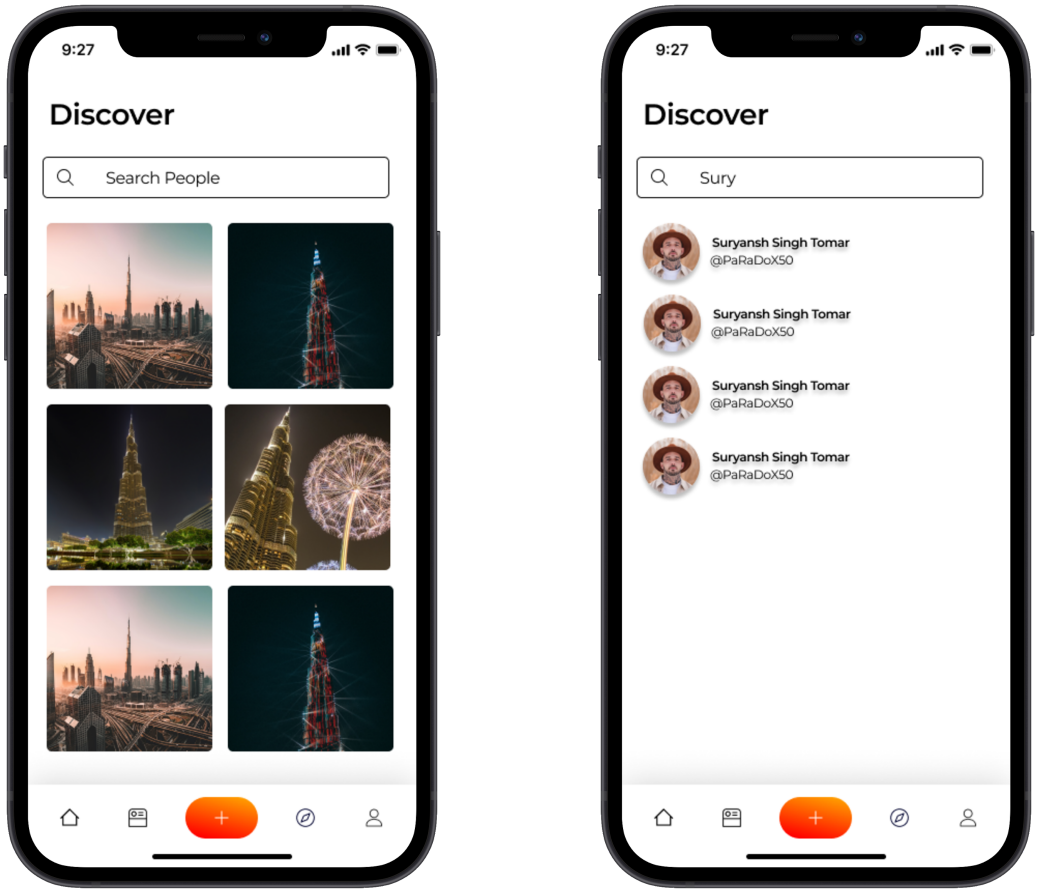
Query to retrieve the notification -

```
_database.collection("users").document("user_id").collection("notifications").getDocuments();
```

This query will retrieve all the notifications. Based on the document's notificationType field, we will differentiate and decide whether the notification is about a like on a post, comment on a post, or any other notification.

We will have to implement lazy loading for this part too.

Discover Screen



This screen will consist of a TextField (Search bar), a ListView (to view the search results) and a GridView (to explore/discover a global feed just like Instagram).

For the user search implementation, we will implement a function inside the onChange parameter of the TextField to search through all the users. The onChange function will be triggered whenever there are changes in the TextField(Search Bar).

Cloud Firestore doesn't support native indexing or search for text fields in documents.

Therefore, to search through all the users, we will have to create an array field named searchParams inside every user document, which will contain all the "Search Parameters" or "Search Strings".

A "Search Parameter" or a "Search String" is a **substring** of a string with the first character, same as the string. For example, "Ale" and "Alex" are two "Search Parameters" for the word "Alexander".

The "Search Parameters" or "Search Strings" can be created using the method below

```
List<String> createSearchParams({String userName, String firstName, String lastName}){
    List<String> searchParams = [ ];
    for (int i = 0; i < userName.length; i++) {
        searchParams.add(userName.toLowerCase().substring(0, i + 1));
    }
    String fullName = "$firstName $lastName".trim();

    for (int i = 0; i < firstName.length + lastName.length + 1; i++) {
        searchParams.add(fullName.toLowerCase().substring(0, i + 1));
    }
    return searchParams;
}
```

This method iterates through the full name as well as the username of the user.

For example, A user with name "Alex Adams" and username "hopsin" will have

```
searchParams = ["a", "al", "ale", "alex", "alexa", "alexad", "alexada",
                "alexadam", "alexadams", "h", "ho", "hop",
                "hops", "hopsi", "hopsin"]
```

To search through all the users, we will just have to retrieve all the user documents whose searchParams field contains the keyword any user typed. For example, if any user types "surya" in the search field, we will find all the documents which contain the keyword "surya" in their searchParams field.

The method to search the users will be something like this:

```
Future<List<User>> searchUser(String keyword) async {
  // Searching documents where searchParams array field contains the keyword user typed.
   QuerySnapshot snapshot = await _database
       .collection("users")   //searching through users collection
       .where("searchParams", arrayContains: keyword.toLowerCase().trim())
       .getDocuments();

   List<User> searchResults = snapshot.documents
       .map((e) => _userFromFirebaseUser(userDocument: e))
       .toList();
   // Converting documents to User data model.
   return searchResults;
 }
```

Here User is a data model.

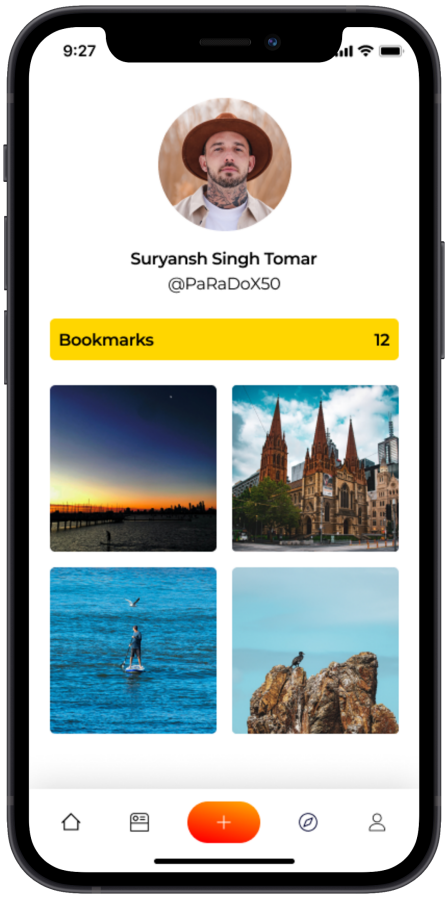
After loading all the results from the database, we will display them in a ListView.

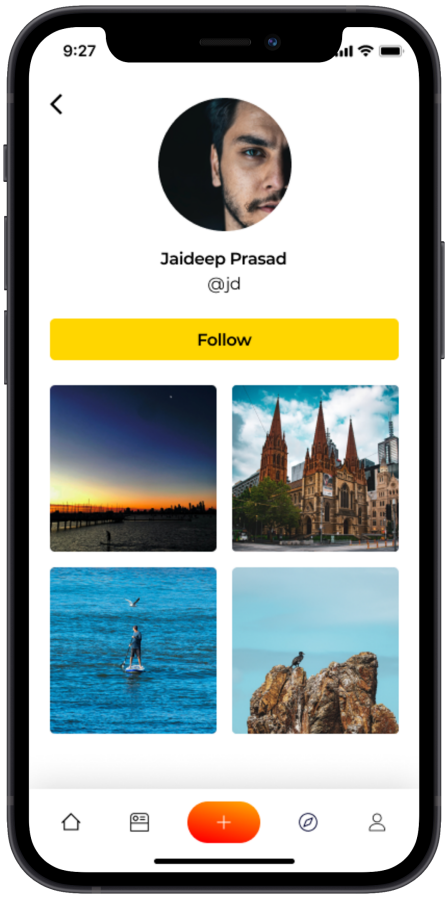
And for the same reasons mentioned in the Feed section, we will have to implement lazy loading in this section as well.

After the results are displayed, clicking on any particular result will take us to their profile. Implementation of Profile screen is discussed in the next section.

And for the global feed part, we retrieve posts from the posts collection randomly.

Profile Screen





This screen will display most of the user's details, like Name, Username, and will display all the posts uploaded by the user.

All the user details will be retrieved using the method below:

```
Future<UserProfile> getUserProfile({String userId}){
    DocumentSnapshot userDoc = await _database.collection("users").document(userId).get();
    return _userProfileFromFirebaseUser(userDoc);
}
```

To retrieve all the posts uploaded by the user, we will use the "where" query and postByUid.

```
Future<List<Post>> getUsersPost({String userId}){
    QuerySnapshot snap = await _database
                                .collection("posts")
                                .where("postByUid",isEqualTo:userId)
                                .orderBy("date")
                                .getDocuments();
    return snap.documents.map(
      (item) => _postFromFirebasePost(item)
    ).toList();
}
```

These posts will be then passed to a ListView.

And for the same reasons mentioned in the Feed section, we will have to implement lazy loading in this section as well.

The profile screen will also have a button to "Follow" the user.

When the "Follow" button is clicked

1. The user-id of the person who clicked is saved inside the SubCollection "followers" of the user document of the person who got followed.
2. The user-id of the person who got followed is saved inside the SubCollection "following" of the user document of the person who followed

```
Future<> followUser({String targetUserId, String currentUserId}) {

  // 1
    await _database.collection("users").document(targetUserId)
        .collection("followers").document(currentUserId)
        .add({"uid":currentUserId});

  // 2
    await _database.collection("users").document(currentUserId)
        .collection("following").document(targetUserId)
        .add({"uid":targetUserId});
  }
```

Comments Screen

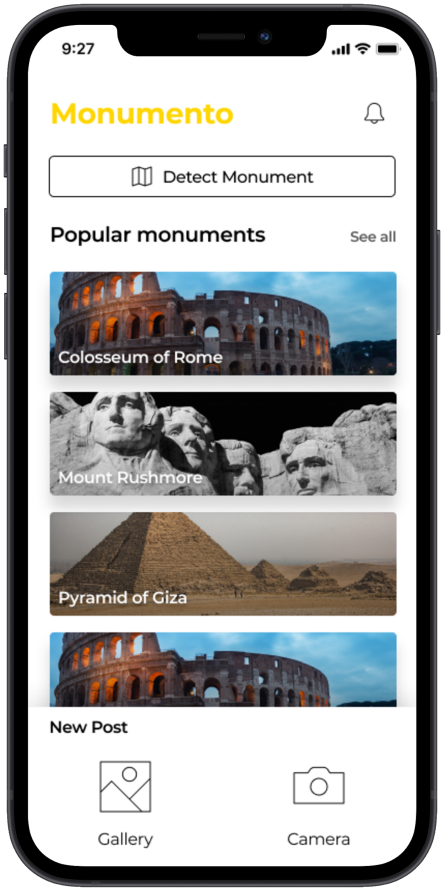
The screen will display all the comments on a post.

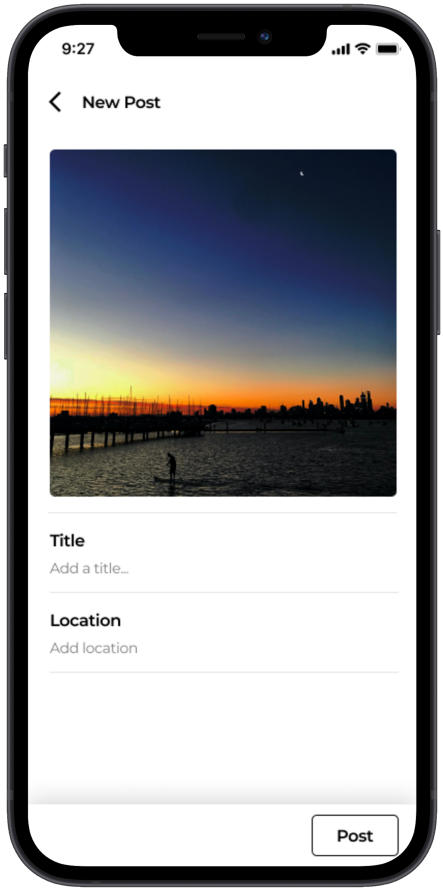
And for the same reasons mentioned in the Feed section, we will have to implement lazy loading in this section as well.

Also, the screen will have a section to add comments.

Whenever a user comments, it will be added to the comments subcollection of the post.

New Post Screen



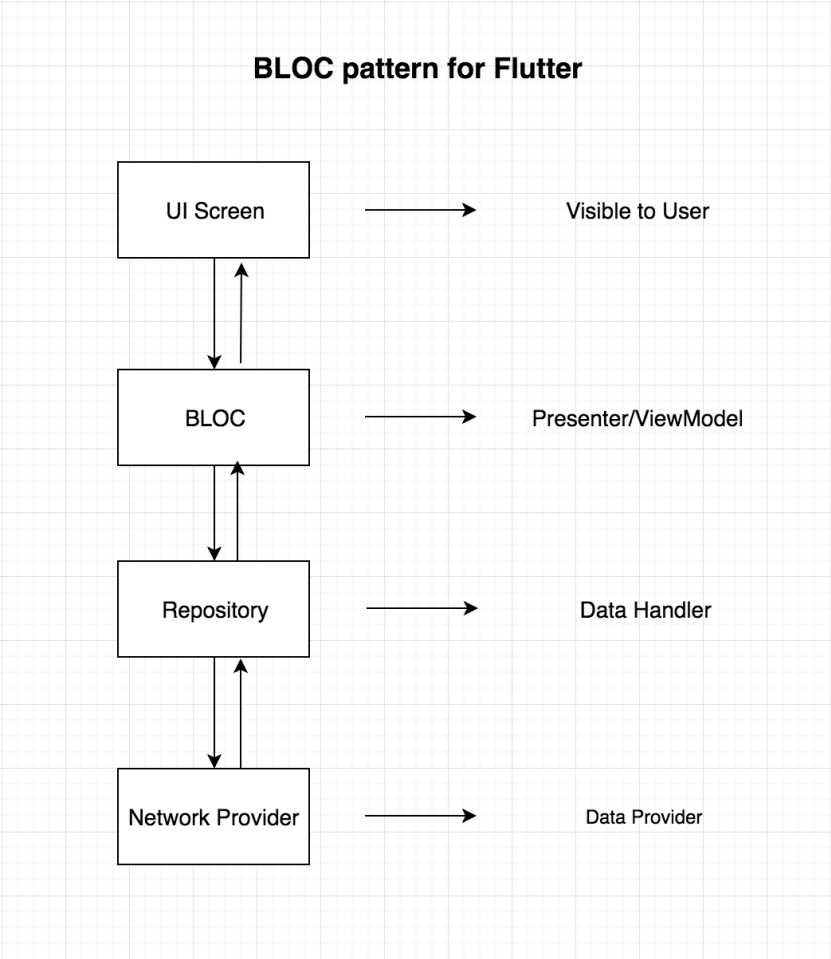


Whenever a user clicks on the "+" button at the centre of BottomNavBar, a bottom sheet with two source options will slide on to the screen. After selecting the image, the user will be redirected to a new screen where he/she will be able to add a title and location to the photograph.
After the "Post" button is clicked, the post will be uploaded and a new post document will be added to the posts collection of the database.

Reference Links

https://firebase.google.com/docs/firestore/best-practices
https://pub.dev/packages/lazy_load_scrollview
https://www.youtube.com/watch?v=Ofux_4c94FI
https://www.youtube.com/watch?v=haMOUb3KVSo

Refactoring the code and restructuring the app into a suitable Pattern/Architecture.



In my opinion, BLoC will be the most suitable choice because it separates the view layer from business logic very well. This entails better reusability and testability.
BLoC pattern has a lot more advantages but the one best suited for Monumento is that it makes migration of the app to a new backend super easy. If in future, we plan to migrate the app to some custom backend, we will only have to edit the repository (data handler) part of the app and will not have to touch any of UI and Business-Logic parts.

Since our app will be more complex after the addition of a new social media section, the emission of separate states with every user interaction will come in very handy.

For example, if the device loses the internet connection, we can emit a state stating that the internet connection is lost and the whole app will then behave accordingly.

I have already migrated the app to BLoC. I have also created a MR for this. Link to the MR monumento!67 .
The only part left is to implement new Entities and Models.

I think BLoC will be a nice choice but I am open to implementing any architecture our mentors suggest this summer.

Reference Links

https://bloclibrary.dev/
https://www.miquido.com/blog/flutter-architecture-provider-vs-bloc/

Timeline

Community Bonding (May 17 - June 7)	• I'll be migrating Native CloudVisionAPI part to Flutter for both Android and iOS. • I'll try to migrate the native android AR implementation to Flutter. Also, I'll try to implement the AR part for iOS in Flutter as well. If everything falls into place, we will be able to proceed further with the Flutter module only. • I'll be discussing with my mentor to get inputs on how I can improve the ideas' implementation. • I'll constantly be adding desired features and fixing bugs mentioned in the Scope of Improvement section as much as I can. • I'll be learning about writing Unit, Widget, and Integration tests because I don't have much experience writing tests for Flutter. But I do know the basics.
Week 1 (June 7 - June 13)	• I'll migrate the app to BLoC architecture. • I'll add Models and Entities for the already implemented features.

Week 2 and Week 3 (June 14 - June 27)	• I'll set up and embed the Flutter Module with a new iOS app. • I'll complete the iOS implementation for all the core features. • The iOS app will be ready to use at the end of this period.
Week 4 and Week 5 (June 28 - July 11)	• I will start implementing screens for the Social Media section • I'll complete the Feed Screen and New Post Screen with every feature I mentioned above in the implementation section.
Week 6 (July 12 - July 16) Mid Evaluations	• At this point, I will be able deliver an application that will be completely compatible with iOS devices and will have a structured and architectured code. • Feed Screen and New Post Screen of Social Media Section will be completely implemented.
Week 7,8,9 (July 17 - August 8)	• I'll complete the rest of the screens, including the Comments Screen, Profile Screen, Search Screen and Notification Screen of the Social Media section.
Week 10 (August 9 - August 15)	• At this time, I will test the application on different devices and try to fix as many bugs as possible.

	• I'll write unit, widget and integration tests for the Flutter Module.
Final Week 11 (August 16 - August 23)	• I'll add documentation and will clean up the application and code for final submission.

My Past Contributions to Monumento

I started contributing to Monumento in the middle of December 2020 and continued to explore and learn.

Issues Opened:

Error : Code not compiling on Flutter 2.0 due to intro_views package
Code refactoring required.
Api : Google is shutting down Poly
Code : 20+ Deprecation and other warnings with the new Flutter 2.0
UI/UX : Incorrect Back button behaviour after Sign Up or Sign In

All the issues opened by me are listed here.

MRs created:

WIP: iOS Implementation
Fixes #48 : Migrated the app to BLoC architectural pattern.
Fixes #41 : Added a contributing.md file
Fixes#43 : Back button behaviour fixed

All the MRs created by me are listed here.

Why are you the best person to execute this proposal?

I have a fair experience in building apps using Flutter, Firebase, Kotlin, Swift and HTTP APIs, which is the required tech-stack for the project. I also know the importance of writing clean and scalable code. I have worked on several projects and with several startups. Therefore, I do have the skillset and experience to execute this project.

Prior Experience

I have been doing App Development for the last one and a half years. I first started developing apps for android using Java.

I worked with Acadza as a Native Android (Kotlin) and React Js Developer Intern. During this Internship, I added Kotlin and React Js to my skill set. I mainly worked with the MVVM architecture, Hilt DI, Coroutines and Room Persistence library during this period.

I was a Native Android Developer Intern for Signo, a Flutter Developer for Droog and a Flutter Developer for one more startup. I mainly worked with Firebase, Flutter Animations, Flutter Architectures during these internships and created an e-Learning Social Media app from scratch.
I also contributed to the development of our Institute's android app.

I have in the past worked with backend frameworks like Django, DRF, Node Js etc.

And from the last 2-3 months, I have been contributing to open-source organisations. During this period, I came to know about the Monumento. And since last month, I am learning Swift and AR to contribute to Monumento and make it support iOS devices.

Post GSoC plans

There are very few chances that some things might go unimplemented because 11 weeks is plenty of time, but still, If it happens, I'll try to complete them post GSoC.

I'll keep contributing to Monumento as much as I can and will keep the development environment running.

Regardless of GSoC, I would love to engage in discussions with the Aossie community to get exposure to new technologies and Ideas. I would love to be of any help even after the GSoC period.