Arjun Dureja May 10, 2020

Prepr – iOS Developer Summer Work Challenge Report

1 Introduction

The following report explains my programming processes when developing an iOS application for the Prepr iOS developer Summer work challenge. The application itself was built entirely in Swift using UIKit, Google Maps API, Google Places API, and Firebase.

2 Programming Processes

I started by coming up with a design for the application. From the requirements given, I knew there had to be a login page, a page showing a list of labs, a page allowing users to create labs, and a page showing a Google maps query for a selected lab. I drew a few mockups and wrote down the core elements I would need to develop this app. The first thing I programmed was the authentication screen, I made it very clean and simple. It has the Prepr logo, two text fields, and a button. I also added a label and button allowing the user to sign up if they do not already have an account. A screenshot of this design can be seen in *Figure 1*.

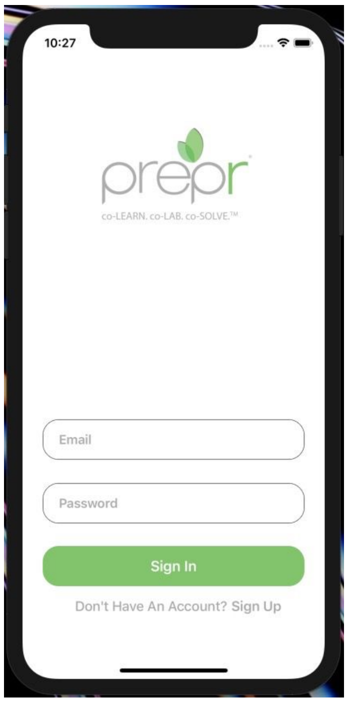


Figure 1. Login Screen

Once the login screen was finished, I began working on the list of labs. I decided to use a UITableView for this as it will allow me to easily manage all the labs, giving functionality for creating, deleting, and editing. In order to set this table view up properly, I needed to create some sort of model. This model would contain all the details for a lab, which would be the name, date added, and location (name and coordinates). I put this into a separate file and added a variable which contained an array of the lab model I just created to the table view controller. Now I could set up my table view by setting "numberOfRowsInSection" to the size of the array. For "cellForRowAt", I came up with a cell design in storyboards and put it into its own file called LabCell, so now "cellForRowAt" would dequeue a reusable cell of type LabCell and return it. The design for this cell can be seen in *Figure 2*. I also called a function called setupCell in "cellForRowAt" that styled the cell and displayed all the lab info. The code for this can be seen in *Figure 3*.

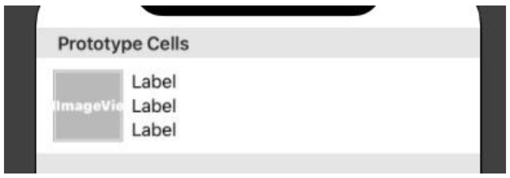
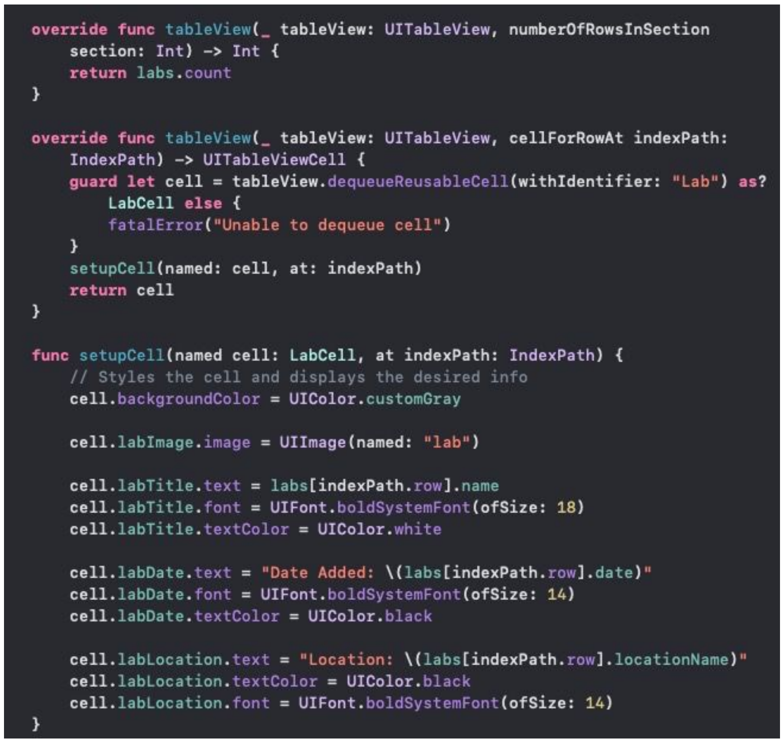


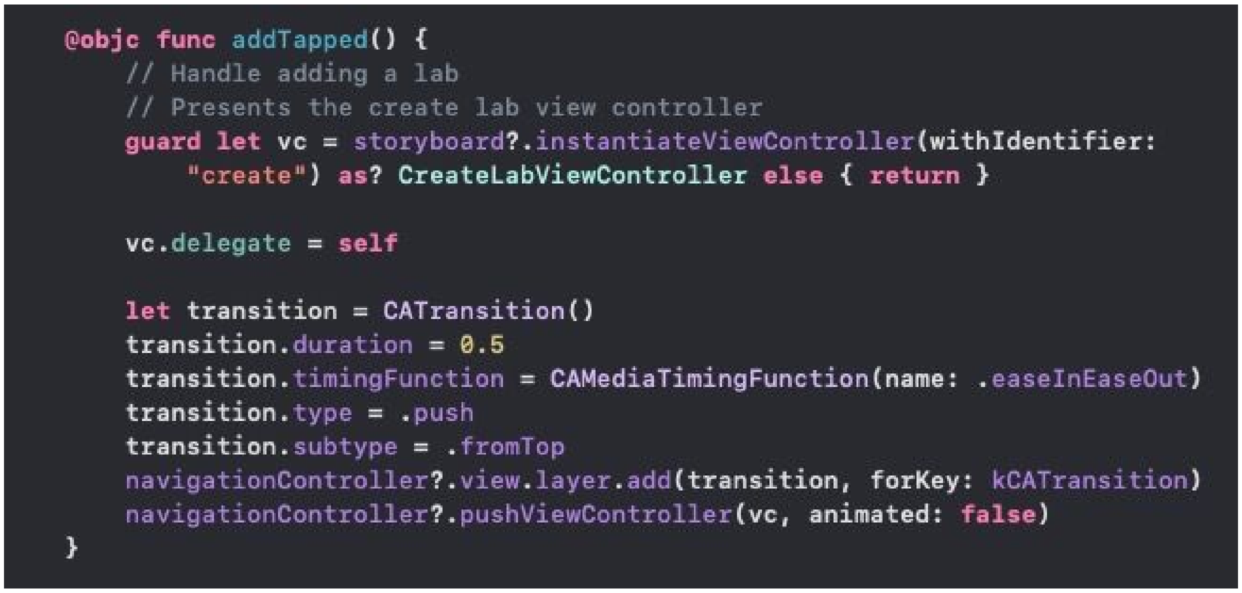
Figure 2. Lab Cell Design



```swift
override func tableView(_ tableView: UITableView, numberOfRowsInSection
    section: Int) -> Int {
    return labs.count
}

override func tableView(_ tableView: UITableView, cellForRowAt indexPath:
    IndexPath) -> UITableViewCell {
    guard let cell = tableView.dequeueReusableCell(withIdentifier: "Lab") as?
        LabCell else {
        fatalError("Unable to dequeue cell")
    }
    setupCell(named: cell, at: indexPath)
    return cell
}

func setupCell(named cell: LabCell, at indexPath: IndexPath) {
    // Styles the cell and displays the desired info
    cell.backgroundColor = UIColor.customGray

    cell.labImage.image = UIImage(named: "lab")

    cell.labTitle.text = labs[indexPath.row].name
    cell.labTitle.font = UIFont.boldSystemFont(ofSize: 18)
    cell.labTitle.textColor = UIColor.white

    cell.labDate.text = "Date Added: \(labs[indexPath.row].date)"
    cell.labDate.font = UIFont.boldSystemFont(ofSize: 14)
    cell.labDate.textColor = UIColor.black

    cell.labLocation.text = "Location: \(labs[indexPath.row].locationName)"
    cell.labLocation.textColor = UIColor.black
    cell.labLocation.font = UIFont.boldSystemFont(ofSize: 14)
}
```

Figure 3 - Table View Code

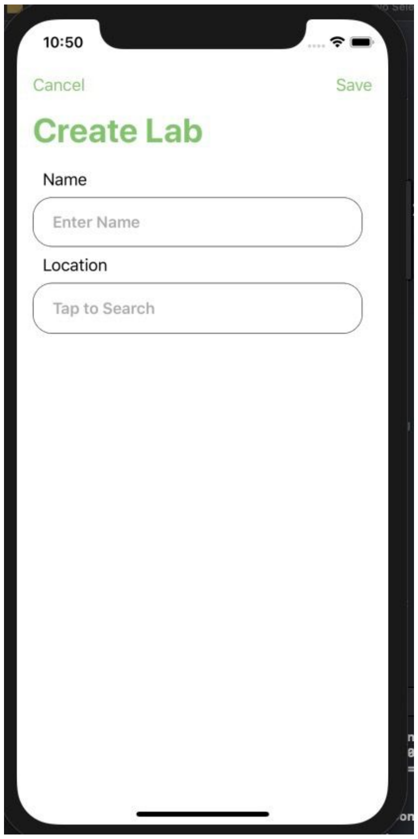
Now I set up a navigation bar button item which the user could tap on to add a new lab. This button would send the user to a different view controller where they could enter in the lab information and save it. I added a selector to the button which instantiated a view controller called “CreateLabViewController”. I also added a transition from the bottom to the top to improve the user experience. The code for this can be seen in *Figure 4.*



```swift
@objc func addTapped() {
    // Handle adding a lab
    // Presents the create lab view controller
    guard let vc = storyboard?.instantiateViewController(withIdentifier:
        "create") as? CreateLabViewController else { return }

    vc.delegate = self

    let transition = CATransition()
    transition.duration = 0.5
    transition.timingFunction = CAMediaTimingFunction(name: .easeInEaseOut)
    transition.type = .push
    transition.subtype = .fromTop
    navigationController?.view.layer.add(transition, forKey: kCATransition)
    navigationController?.pushViewController(vc, animated: false)
}
```

Figure 4. Function Called When User Taps “Add” Button

I designed the create lab view controller using storyboards. It contains two text fields, two labels, and a button. The text fields allowed the user to enter the name of the lab, and the location. A screenshot of this screen can be seen in *Figure 5.* For the location, I decided to use Google Places API which has something called an “autocomplete controller” which allows users to search for a location and the API will automatically show the users all the locations that match their query. This was quite easy to implement, Google’s documentation was very clear and I had no issues with setting it up. In order to show the user the autocomplete view controller, I placed an invisible button on top of the location textfield, so that when the user taps it, it triggers an action to open the autocomplete view controller. A screenshot of this can be seen in *Figure 6*. Once the user finishes entering a location, the text field is updated and they are able to press the save button.



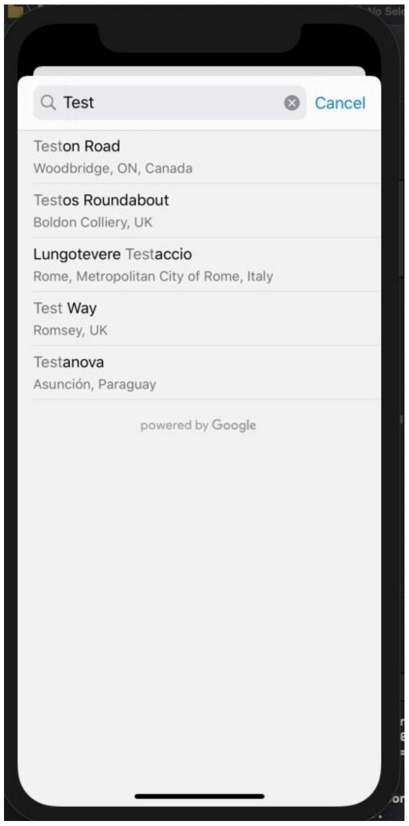
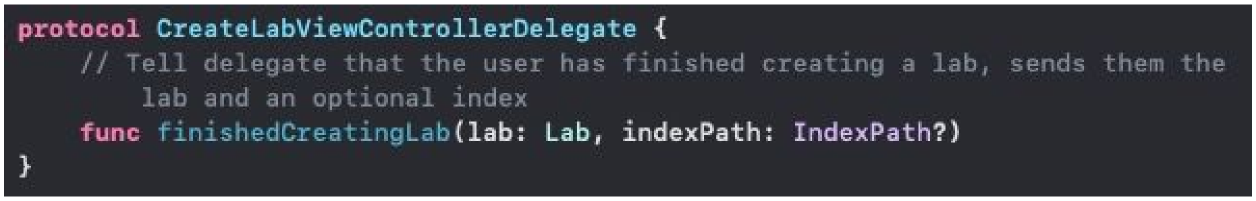


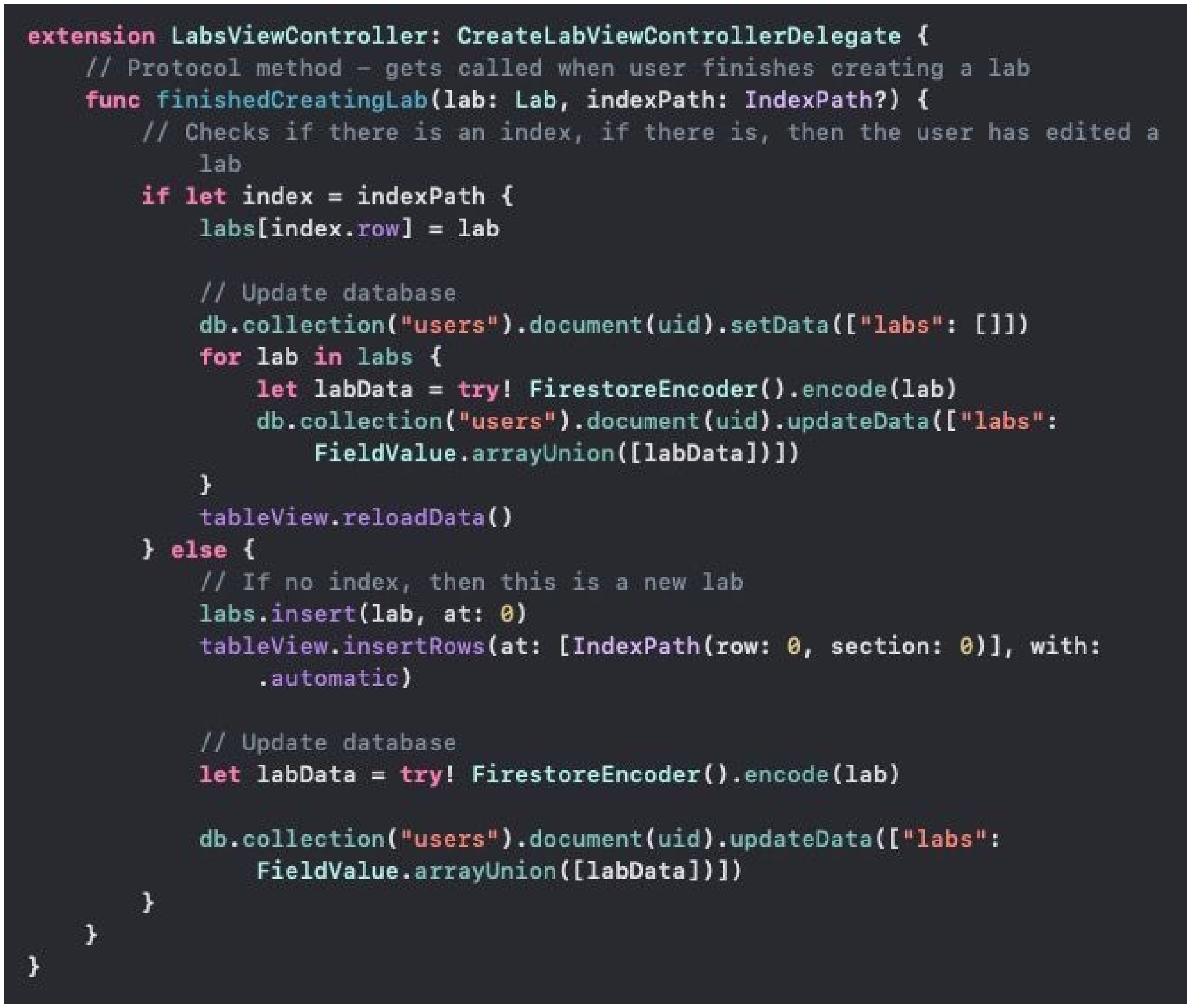
Figure 5. Create Lab Screen *Figure 6. Google Places Autocomplete Controller*

Now arises the first major challenge, I needed a way to send the data from the create lab view controller, to the lab view controller which displays the table view of labs. Once the user presses save, I wanted the table view to update immediately with all the information. I came up with a few ways to solve this problem and decided to use the protocol and delegate communication pattern. This allows me to tell the lab view controller that the user has finished creating a lab, and then send that data to it. I created a protocol called “CreateLabViewControllerDelegate” and added a function called “finishedCreatingLab” which took a Lab object and an optional index as parameters. The reason I added the index was for when users needed to edit labs. A screenshot of this protocol can be seen in *Figure 7.* I added a variable of this protocol in the create lab view controller and named it “delegate”. Then I added a call to the “finishedCreatingLab” function when the user presses the save button, which sends in the lab and the optional index. Now I needed to set up the delegate, which was the lab view controller. So I added an extension to the lab view controller which conforms to the create lab view controller delegate protocol. Since it conformed to the protocol, it had to implement the “finishedCreatingLab” function which is called when the user presses save. Now I have all the data I need to add to the lab list. So I checked if the user is creating a new lab or editing, and added the lab to the lab array and table view. A screenshot of this extension can be seen in *Figure 8*.



```swift
protocol CreateLabViewControllerDelegate {
    // Tell delegate that the user has finished creating a lab, sends them the
        lab and an optional index
    func finishedCreatingLab(lab: Lab, indexPath: IndexPath?)
}
```

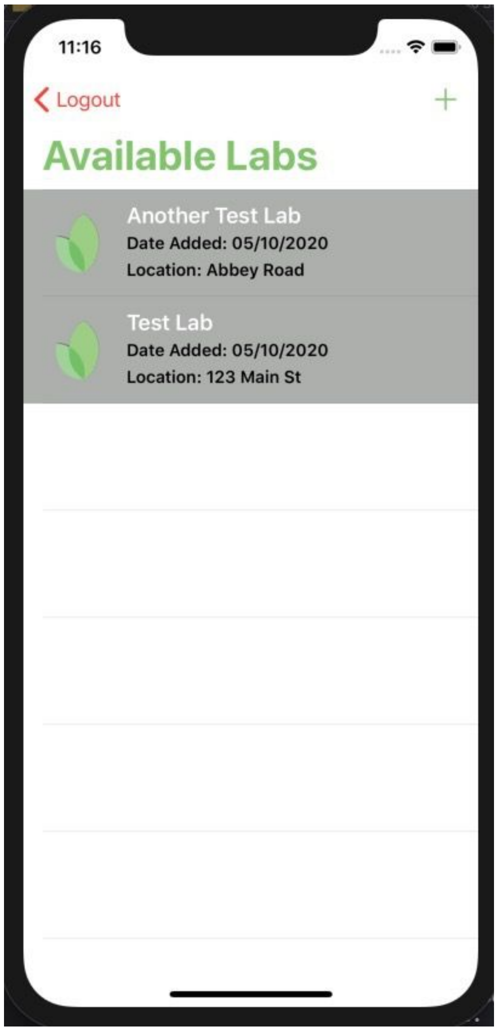
Figure 7. CreateLabViewControllerDelegate Protocol



```swift
extension LabsViewController: CreateLabViewControllerDelegate {
    // Protocol method - gets called when user finishes creating a lab
    func finishedCreatingLab(lab: Lab, indexPath: IndexPath?) {
        // Checks if there is an index, if there is, then the user has edited a
            lab
        if let index = indexPath {
            labs[index.row] = lab

            // Update database
            db.collection("users").document(uid).setData(["labs": []])
            for lab in labs {
                let labData = try! FirestoreEncoder().encode(lab)
                db.collection("users").document(uid).updateData(["labs":
                    FieldValue.arrayUnion([labData])])
            }
            tableView.reloadData()
        } else {
            // If no index, then this is a new lab
            labs.insert(lab, at: 0)
            tableView.insertRows(at: [IndexPath(row: 0, section: 0)], with:
                .automatic)

            // Update database
            let labData = try! FirestoreEncoder().encode(lab)

            db.collection("users").document(uid).updateData(["labs":
                FieldValue.arrayUnion([labData])])
        }
    }
}
```

Figure 8. LabsViewController Extension Which Conforms to the Protocol in Figure 7.

Now I had a way for both view controllers to communicate which allowed users to create labs. A screenshot of the labs tableview with a couple example labs can be seen in *Figure 9.* The next step was to allow the user to tap on labs they've added. So I implemented the "didSelectRowAt" table view method which I made to display a UIAlertController as an action sheet. This action sheet would have four buttons, "View Location", "Edit Details", "Delete", and "Cancel". Starting with view location, I made this button navigate to a new view controller that showed the location as a Google Maps Query. I needed to use the Google Maps API for this which was also quite simple to set up. I added a marker for the location and zoomed in on it. A screenshot of this can be seen in *Figure 10*.



Available Labs

Another Test Lab
Date Added: 05/10/2020
Location: Abbey Road

Test Lab
Date Added: 05/10/2020
Location: 123 Main St

Figure 9. Labs Table View With Example Labs Controller

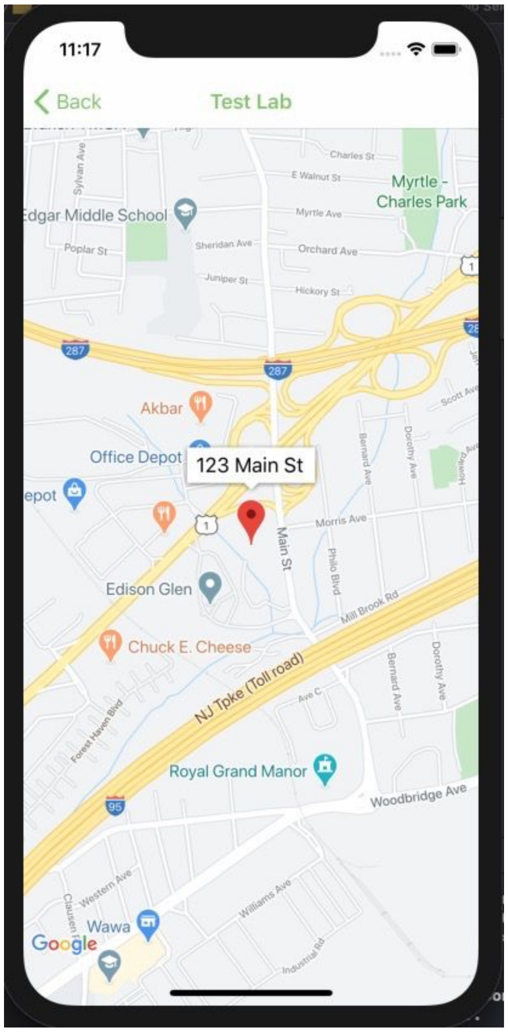
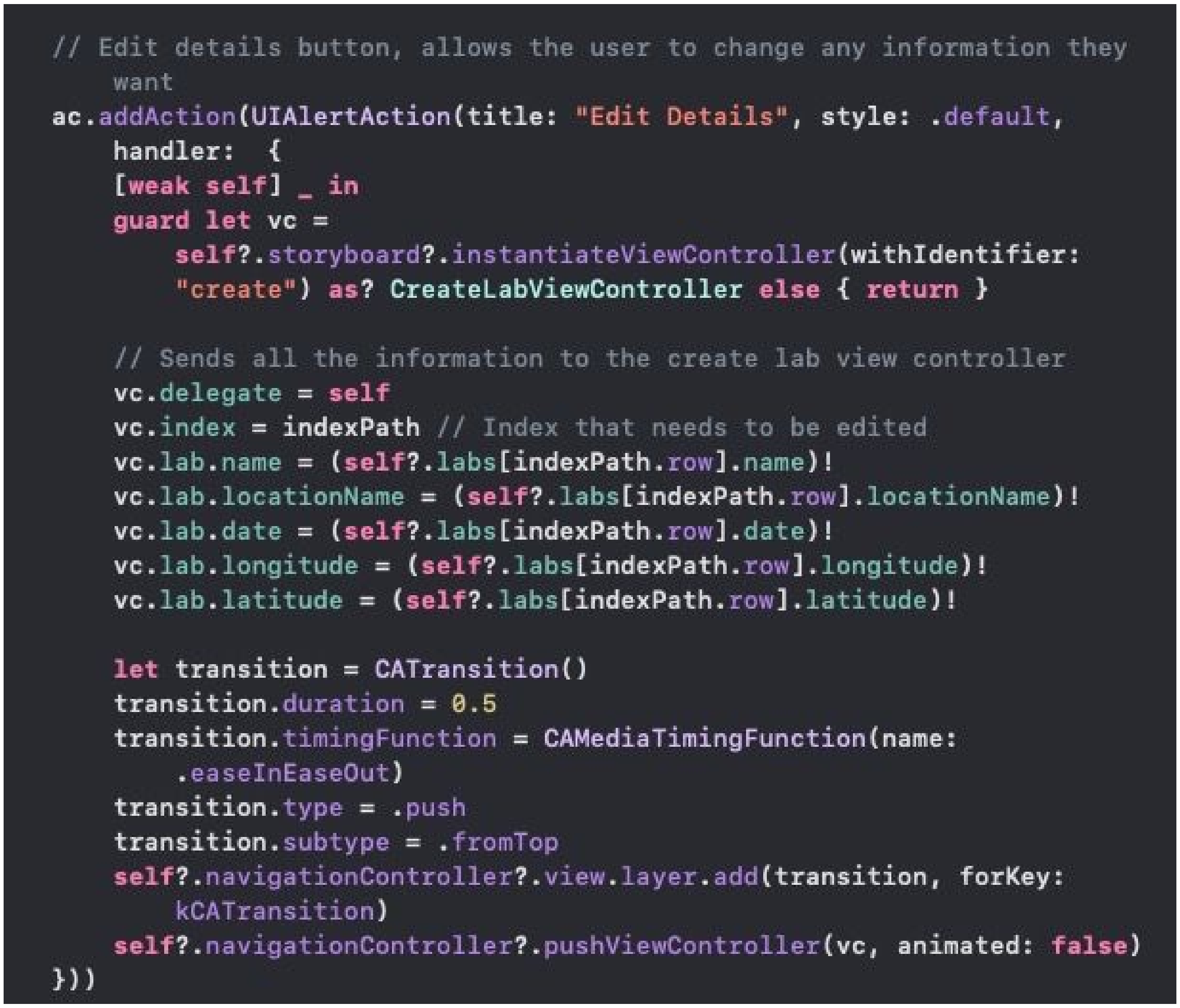


Figure 10. Map View

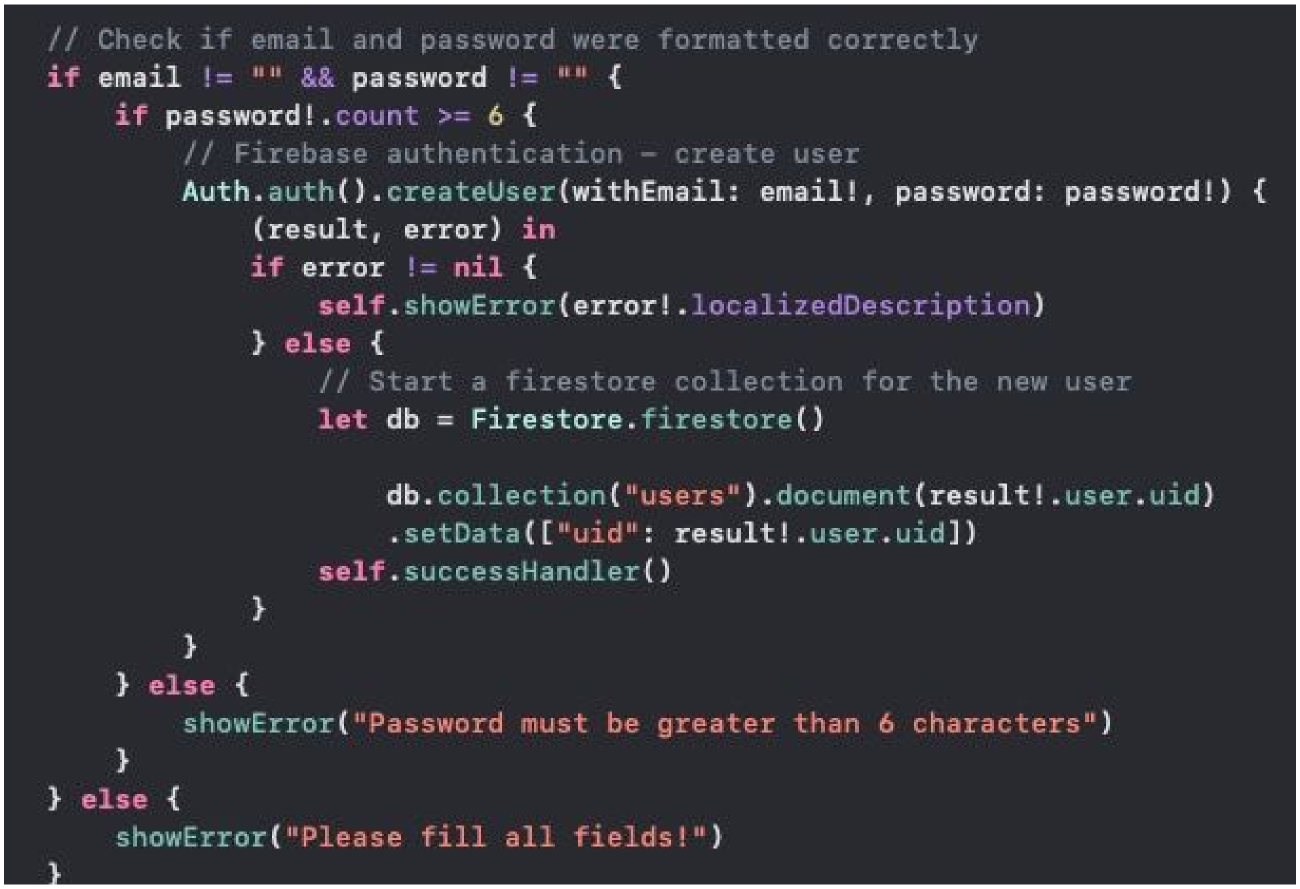
The next button on the action sheet was “Edit Details” which brought up the Create Lab view controller but instead of creating a new lab, it edited an existing one. I did this by filling in all the information before showing the view controller and sending in the index path of the cell the user wants to edit. This way, I could use the same protocol as before to communicate between views, and just edit the cell instead of creating a new one since the optional index is not nil. A screenshot of the edit details UIAlertAction can be seen in *Figure 11*. The next two buttons were easy to implement, “Delete” just removed the cell from the array and table view. “Cancel” just closed the action sheet and de-selected the cell.



```
// Edit details button, allows the user to change any information they
    want
ac.addAction(UIAlertAction(title: "Edit Details", style: .default,
    handler:  {
    [weak self] _ in
    guard let vc =
        self?.storyboard?.instantiateViewController(withIdentifier:
        "create") as? CreateLabViewController else { return }

    // Sends all the information to the create lab view controller
    vc.delegate = self
    vc.index = indexPath // Index that needs to be edited
    vc.lab.name = (self?.labs[indexPath.row].name)!
    vc.lab.locationName = (self?.labs[indexPath.row].locationName)!
    vc.lab.date = (self?.labs[indexPath.row].date)!
    vc.lab.longitude = (self?.labs[indexPath.row].longitude)!
    vc.lab.latitude = (self?.labs[indexPath.row].latitude)!

    let transition = CATransition()
    transition.duration = 0.5
    transition.timingFunction = CAMediaTimingFunction(name:
        .easeInEaseOut)
    transition.type = .push
    transition.subtype = .fromTop
    self?.navigationController?.view.layer.add(transition, forKey:
        kCATransition)
    self?.navigationController?.pushViewController(vc, animated: false)
}))
```

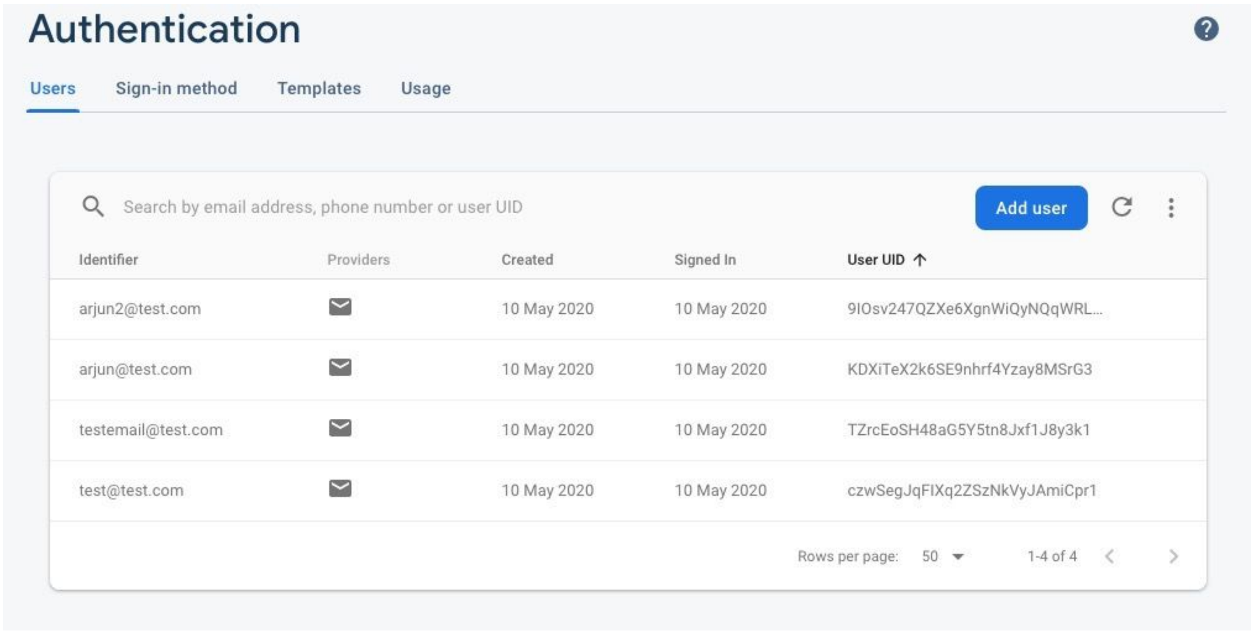
Figure 11. Edit Details UIAlertAction

The frontend of the app is now nearly finished. I have successfully implemented CRUD and added all the necessary screens. Now I need to build a backend, which is a challenge on its own. I decided to build this by using Firebase as I have used it before and they have great support and documentation for iOS. I needed to use two Firebase functionalities: authentication, and Firestore. I realized that in order for this to work properly, I needed to create a signup screen. So I quickly came up with a design and added code to show the screen when the user taps the “Signup” button. A screenshot of this can be seen in *Figure 12*. This was my first time setting up authentication, so I read through all the documentation and started to code. I decided to use the email/password method of authentication as it is the most common. Firebase has an Auth object you can use to authenticate and create users. So I called an instance of this object to whatever the user entered in the email and password fields on the sign up page. This would check if they entered valid information, such as a long enough password, valid email address format, or if the email already exists in the database. Once the user is successfully created, they are added to the database. A screenshot of this with example users can be seen in *Figure 13*.



```
// Check if email and password were formatted correctly
if email != "" && password != "" {
    if password!.count >= 6 {
        // Firebase authentication - create user
        Auth.auth().createUser(withEmail: email!, password: password!) {
            (result, error) in
            if error != nil {
                self.showError(error!.localizedDescription)
            } else {
                // Start a firestore collection for the new user
                let db = Firestore.firestore()

                    db.collection("users").document(result!.user.uid)
                    .setData(["uid": result!.user.uid])
                self.successHandler()
            }
        }
    } else {
        showError("Password must be greater than 6 characters")
    }
} else {
    showError("Please fill all fields!")
}
```

Figure 12. Code to Create a New User Using Firebase



Authentication

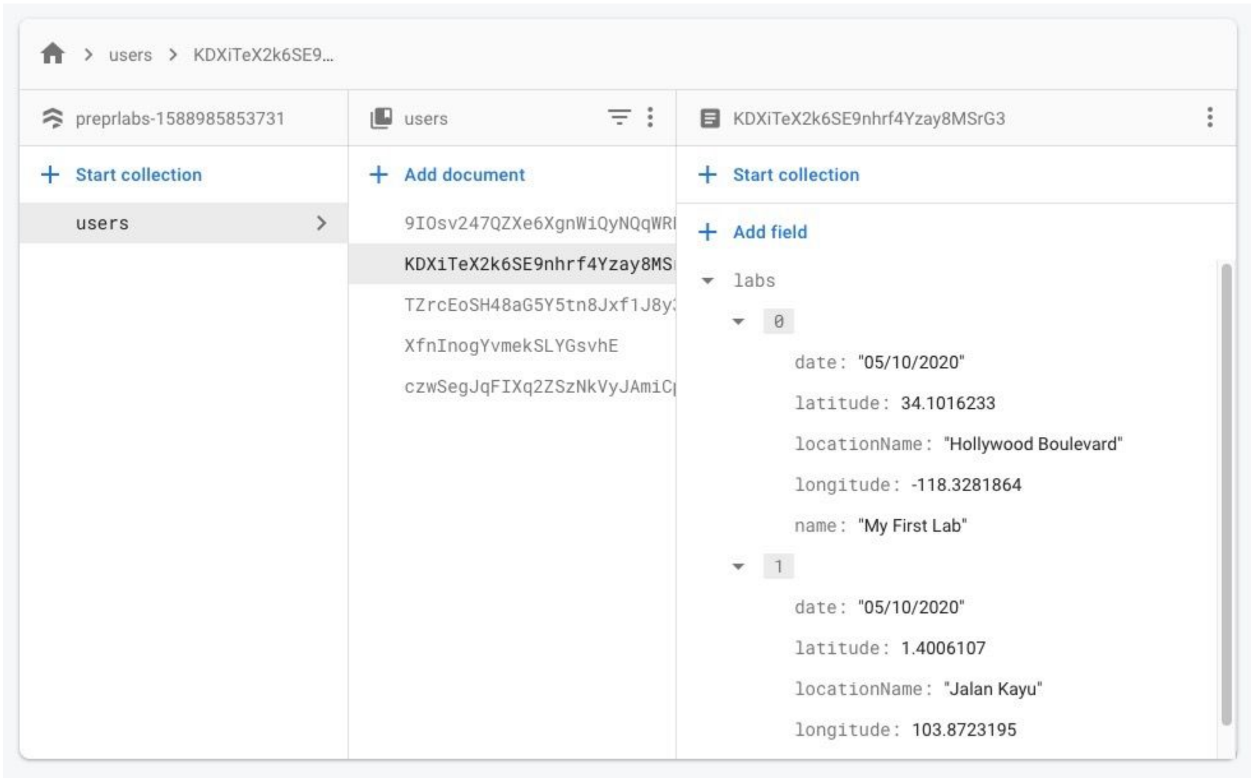
Identifier	Providers	Created	Signed In	User UID ↑
arjun2@test.com	✉	10 May 2020	10 May 2020	9IOsv247QZXe6XgnWiQyNQqWRL...
arjun@test.com	✉	10 May 2020	10 May 2020	KDXiTeX2k6SE9nhrf4Yzay8MSrG3
testemail@test.com	✉	10 May 2020	10 May 2020	TZrcEoSH48aG5Y5tn8Jxf1J8y3k1
test@test.com	✉	10 May 2020	10 May 2020	czwSegJqFIXq2ZSzNkVyJAmiCpr1

Rows per page: 50 1-4 of 4

Figure 13. Firebase Authentication Database

This database allowed me to view all the current users, delete them, and reset their passwords. After the user signs up, they are brought back to the login page where they can enter their credentials. This works similarly to the signup screen but instead of calling the “createUser” method, I would call the “signIn” method which checks the database if the user exists. Authentication is now finished and is fully working. Now I needed to store user data into Firestore so that the labs they create don’t go away when they close the app. In my opinion, this was the biggest challenge in creating the app. I needed to store quite a bit of information and assign it to the corresponding user. I started by finding a way to differentiate between users. Firebase makes this quite easy as each user has a unique ID attached to it, as you can see in *Figure 13*. So I sent this unique ID to the labs view controller so that it can know which user is currently signed in. I needed to send lab data to Firebase whenever a user creates a lab, updates a lab, or deletes a lab. I also needed to get data from Firebase when the lab view controller loads so that it can populate the table view.

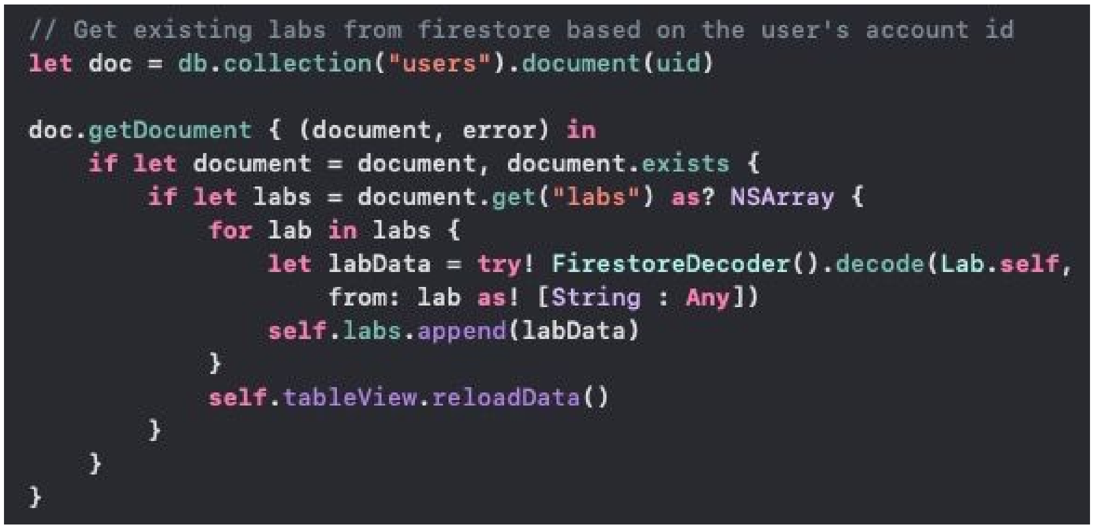
Since I was already familiar with Apple’s “Codable” library which allows you to save data locally in JSON, I decided to use a library called CodableFirebase which uses the same methodologies as Codable but works with Firebase. This made saving all the data much easier as I could encode everything, and then decode it when I needed to access it. If you take a look back at *Figure 8*, you can see the code I wrote to save the data. If the user is creating a new lab, the data is sent to Firebase as an array inside of a document with the name of their unique user ID. The same happens when the user edits a lab but instead of saving as a new entry, it modifies the current entries. A screenshot of the database can be seen in *Figure 14*.



```
labs
  0
    date: "05/10/2020"
    latitude: 34.1016233
    locationName: "Hollywood Boulevard"
    longitude: -118.3281864
    name: "My First Lab"
  1
    date: "05/10/2020"
    latitude: 1.4006107
    locationName: "Jalan Kayu"
    longitude: 103.8723195
```

Figure 14. Firestore Database

It starts with a collection of “users” which holds documents by the name of user IDs. These documents hold arrays of labs which hold all the information required to populate the table view. The code written to get all the user data can be seen in *Figure 15*.



```swift
// Get existing labs from firestore based on the user's account id
let doc = db.collection("users").document(uid)

doc.getDocument { (document, error) in
    if let document = document, document.exists {
        if let labs = document.get("labs") as? NSArray {
            for lab in labs {
                let labData = try! FirestoreDecoder().decode(Lab.self,
                    from: lab as! [String : Any])
                self.labs.append(labData)
            }
            self.tableView.reloadData()
        }
    }
}
```

Figure 15. Getting Data from Firestore

The app is now complete. It has a fully functional frontend and backend. Users can sign up, sign in, create, read, update, and delete labs. They can also view the locations of all of their labs and create multiple accounts.

3 Conclusion

I learned a lot while developing this app and really enjoyed it. Some of the things I learned were: communicating between views efficiently, creating complex table views, using Google Maps and Places API, setting up Firebase authentication, storing user specific data in Firestore, and a lot more. Overall, I believe it went very well, I got everything working and even though I faced many challenges, I was able to solve all of them. Thank you Prepr for giving me the opportunity to complete this challenge. Hope to talk to you soon!