CPSC 290 Final Writeup

Neal Soni, in collaboration with Dylan Gleicher

Table of Contents

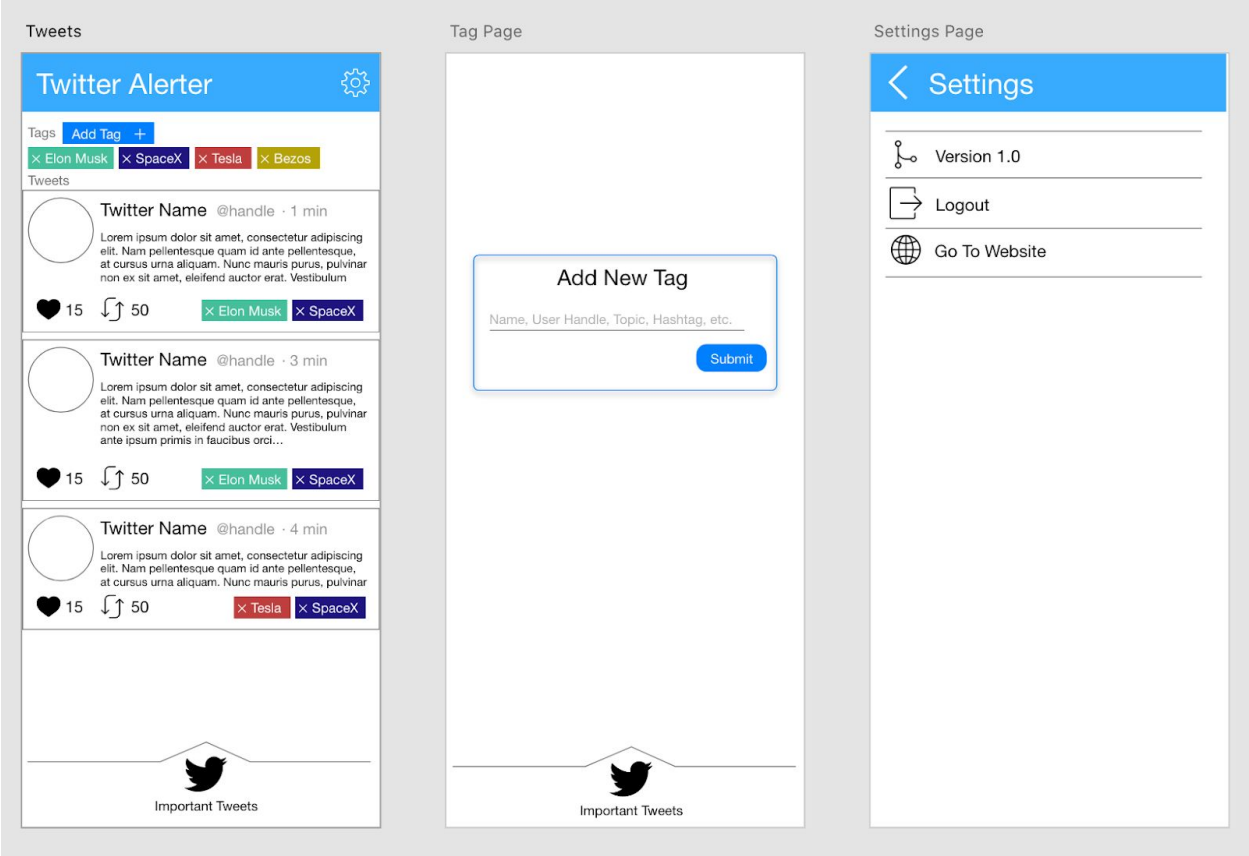
Section 1: Abstract & Introduction

The goal of this independent study was to learn the tools necessary to build a production mobile application, using scalable server infrastructure, and testable code style. The project we determined to allow for us to learn all these principles was a TwitterAlerter. TwitterAlerter is meant to notify a user of a specific tweet that is of high importance to their investments — such as Elon Musk tweeting “considering taking Tesla private at $420. Funding secured” in August 2018. The reason for this is because it is a relatively straightforward architecture, requiring not too many moving parts at any one step, and could be easily separated between the server and mobile app code, allowing for independent development. Throughout this process, alot was learned on the principles of user interface design, Model-View-Controller code design pattern, creating reusable auto-completable network layers under the dependency injection framework, and writing testing code at every level of the process, allowing us to test the models, views, controllers, and networking layer. Additionally, we learned the process of building and deploying these applications in production settings using quality assurance testing methods, testing locally, then in staging, then in production while releasing to different AWS servers and databases. The whole process was extremely rewarding as we learned every step of the app development process from idea to release.

The following sections describe this process of developing this application, from initial design, to storyboarding, to creating testable network layers, and finally testing every layer of the application.

Section 2: Initial Project Design

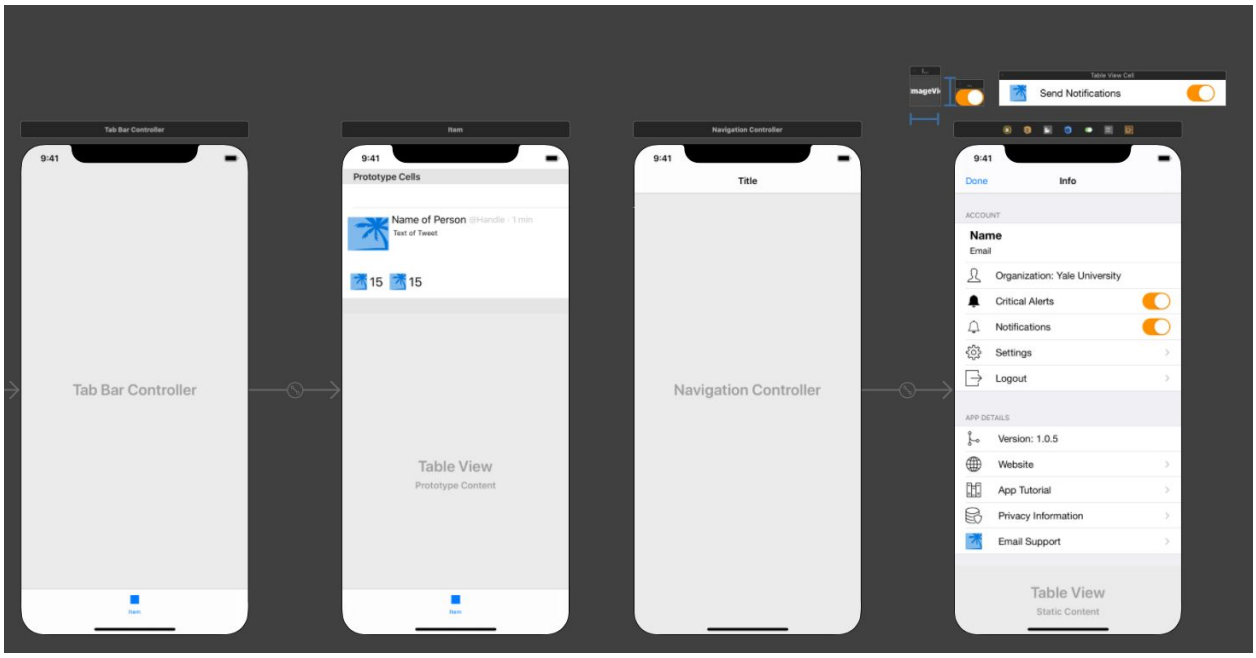
Adobe XD User Interface Design Process



We designed the initial interface in Adobe XD to allow for cross-platform compatibility. This is what we determined an interface for the app would look like because it gives us the raw aspects we need to test different production methods and styles, such as dependency injection, unit tests, user interface (UI) tests, and receiving critical alerts from network providers such as OneSignal.

Using XD was super insightful because it actually parallels the constraint system that both iOS and Android use when creating interfaces, however, it is a much more stripped down version. We learned the containing process for XD and how to simulate a fake user experience and flow between storyboards.

Conversion to iOS Storyboard



The containing of the views was the most challenging part. Since we don't know how much text is going to be in any one tweet, we don't know how much text the one cell and each of it's UI labels is going to have to contain. I found a unique hack that when you constrain the text cell with specific properties, such as using ≥ constraints, I can add this to the table view for that view to make the cell automatically dimension its own height.

```
// Automatic row heights when there is extra data
override func tableView(_ tableView: UITableView, heightForRowAt indexPath: IndexPath) -> CGFloat {
    return UITableView.automaticDimension
}
```

Connecting this table view to the code was a challenge in itself. The 'identifiers' for the cells are extremely finicky and the project had to be rebuilt and refactored a couple times to adhere to the requirements of iOS and Xcode.

Hooking up the views also has its own challenges if you want to rename variables and put in images from online. To make the image fetching and caching easier, we used a library called Kingfisher that automatically stores downloaded images in local app storage and makes it relatively simple to inject these binaries into the ImageViews. It does this by creating an extension to UIImageView with a .kf property allowing it to inject information into the parent, such as loading animations and default binaries.

Section 3: iOS Project

Most of the iOS development was focused on making code that was up to industry standards as possible, using a variety of sources from online, popular github repositories, and consultation with Professor Brown. Below is a description on how we structured the iOS code to be testable, and added a variety of different testing mechanisms.

Network Manager

The network layer of this application is written in Moya. Moya simplified the whole network requesting process from the original version written custom in Alamofire. Moya is built on top Alamofire but improves it in many ways: Allows for autocomplete, easier differentiation between success, error, and failure network requests, makes code more testable both through mocked data apis and through dependency injection into networkable view controllers. Also the code makes it super easy and readable to add network requests on top of an existing api and post process the received json into objects using Decoder.

```
// MARK: - TwitterReporterAPI TargetType Protocol Implementation
extension TwitterReporterAPI: TargetType {
    var enviormentBaseURL : String {
        switch NetworkManager.environment {
        case .Production: return "https://api.TwitterReporter.herokuapp.com/"
        case .Staging: return "https://staging.TwitterReporter.herokuapp.com/"
        case .Dev: return "https://dev.TwitterReporter.herokuapp.com/"
        }
    }

    var baseURL: URL {
        guard let url = URL(string: enviormentBaseURL) else { fatalError("base URL
            could not be determined")
        }
        return url
    }

    var path: String {
        switch self {
        case .getTweets:
            return "/tweets"
        }
    }
    var method: Moya.Method {
        switch self {
        case .getTweets:
            return .get
        }
    }

    var sampleData: Data {
        switch self {
        case .getTweets:
            return getFakeData()
        }
    }

    var task: Task {
        switch self {
        case .getTweets:
            return .requestPlain
        default:
            return .requestPlain
        }
    }

    var headers: [String: String]? {
        return ["Content-Type": "application/json"]
    }
}
```

```
protocol Networkable {
    var provider: MoyaProvider<TwitterReporterAPI> { get }
    func getTweets(completion: @escaping ([Tweet])->())
//    var networkProvider: NetworkManager! { get set }
}


class NetworkManager: Networkable {
    static let environment: Environment = .Production
    static let TwitterReporterAPIKey = "APIKEY"
    var provider = MoyaProvider<TwitterReporterAPI>(stubClosure:
        MoyaProvider.immediatelyStub, plugins: [NetworkLoggerPlugin()])

    func getTweets(completion: @escaping ([Tweet])->()) {
        provider.request(.getTweets) { result in
            switch result {
            case let .success(response):
                do {
                    let results = try JSONDecoder().decode([Tweet].self, from:
                        response.data)
                    completion(results)
                } catch let err {
                    print(err)
                }
            case let .failure(error):
                print(error)
            }
        }
    }
}
```

Left: The original Moya networking layer with the single network request: getTweets. Additionally you can see the differentiation between the staging, dev, and production server connections. We added another tier called ‘local’ so that when we are connected to a local server to test new apis and routes without having to deploy to our amazon staging server.

Right: This is the Network Manager that is instantiated in the beginning of the project and is passed on initialization into every view controller. This makes it super easy to switch between

actual network requests and fake data models stored in the code — more on that later (note the ‘stubClosure: MoyaProvider.immediatelyStub’) in the provider initialization.

This framework for structuring the project is super simple and easy to replicate going forward with any api endpoints, as all one has to do to connect it is add a few simple lines of code to the left view and a standardized method on the right side. The data processing and parsing is all done for you.

Data Model

```
struct Tweet {
    let profileImageURL: String
    let nameOfPerson: String
    let handle: String
    let time: String
    let textOfTweet: String
    let likesCount: String
    let retweetCount: String
    let tag: String
}

extension Tweet: Decodable {
    enum TweetCodingKeys: String, CodingKey {
        case profileImageURL = "profile_image_url"
        case nameOfPerson = "author"
        case handle
        case time
        case textOfTweet = "text_of_tweet"
        case likesCount = "likes_count"
        case retweetCount = "retweet_count"
        case tag
    }
    init(from decoder: Decoder) throws {
        let container = try decoder.container(keyedBy: TweetCodingKeys.self)
        profileImageURL = try container.decode(String.self, forKey: .profileImageURL)
        nameOfPerson = try container.decode(String.self, forKey: .nameOfPerson)
        handle = try container.decode(String.self, forKey: .handle)
        time = try container.decode(String.self, forKey: .time)
        textOfTweet = try container.decode(String.self, forKey: .textOfTweet)
        likesCount = try container.decode(String.self, forKey: .likesCount)
        retweetCount = try container.decode(String.self, forKey: .retweetCount)
        tag = try container.decode(String.self, forKey: .tag)
    }
}
```

Above is a sample data model we made for a Tweet object passed from the server. This model automatically decodes the raw input JSON (shown later) into the appropriate tweet information to reference in the view controller and UITableViewCell code. Additionally, these can be processed as [Tweet].self objects so that when we fetch a list from the server, they are automatically put into an array of tweet objects — and thus much more easily displayed using an iterative loop through inside the UITableView.

Receive One-Signal Critical Alerts

```
override func didReceive(_ request: UNNotificationRequest, withContentHandler contentHandler: @escaping (UNNotificationContent) -> Void) {
    self.receivedRequest = request
    self.contentHandler = contentHandler
    bestAttemptContent = (request.content.mutableCopy() as? UNMutableNotificationContent)

    sendReadReceipt(notificationId: request.identifier)

    if let bestAttemptContent = bestAttemptContent {
          let deliveryTypeString = payload.additionalData[AppDelegate.DeliveryTypeKey] as? String,
          let deliveryType = DeliveryType.fromKey(deliveryTypeString),

        if
            let customContent = bestAttemptContent.userInfo["custom"] as? [AnyHashable: Any],
            let additionalInfo = customContent["a"] as? [AnyHashable: Any],
            let deliveryType = additionalInfo["delivery_type"] as? String
        {
            switch deliveryType {
            case "silent":
                // Turn off the sound
                // TODO: Should we still let it vibrate?
                bestAttemptContent.sound = nil
            case "default":
                  bestAttemptContent.sound = UNNotificationSound.de
                break
            case "emergency":
                // Apply the critical alert
                if #available(iOSApplicationExtension 12.0, *) {
                    bestAttemptContent.sound = UNNotificationSound.defaultCriticalSound(withAudioVolume: 1.0)
                      UNNotificationSound.criticalSoundNamed(name: UNNotificationSoundName, withAudioVolume: Float)
                } else {
                    // TODO: How do we fallback critical alerts?
                }
            default:
                break
            }
        }

        OneSignal.didReceiveNotificationExtensionRequest(self.receivedRequest, with: self.bestAttemptContent)
        contentHandler(bestAttemptContent)
    }
}
```

One of the requirements we set for ourselves was to allow the user to receive ‘critical alerts’ — these are alerts that notify the user that there is an issue that requires their immediate attention, such as the elon 420 tweet. These high property tweets can be sent using this critical property to override the user’s ringer, ensuring they acknowledge the tweet’s existence and take appropriate action. Getting approval for this permission took a couple of weeks and working with an apple representative to make it happen.

Network Testing with Moya

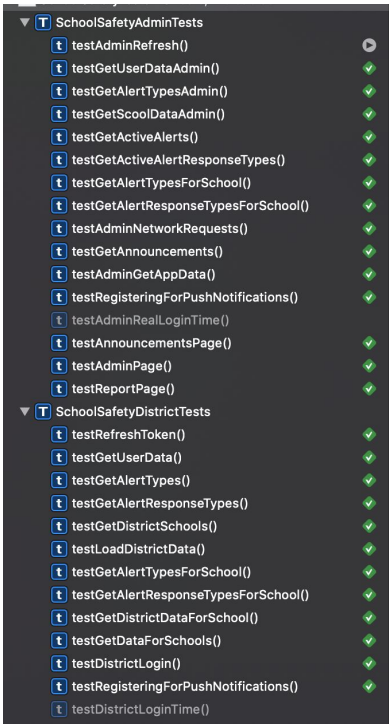
```swift
//
//  File.swift
//  TwitterReporteriOS
//
//  Created by Neal Soni on 2/30/20.
//  Copyright © 2020 Yale University. All rights reserved.
//

import Foundation

func getFakeData() -> Data {
    return stubbedResponse("faketweets")
}

// MARK: - Provider support

func stubbedResponse(_ filename: String) -> Data! {
    @objc class TestClass: NSObject { }

    let bundle = Bundle(for: TestClass.self)
    let path = bundle.path(forResource: filename, ofType: "json")
    return (try? Data(contentsOf: URL(fileURLWithPath: path!)))
}
```

```json
[{
    "profile_image_url" :
        "https://pbs.twimg.com/profile_images/1255961776035729408/0AjDFNaf_400x400.jpg",
    "author" : "Elon Musk",
    "handle" : "@elonmusk",
    "time" : "1 min",
    "text_of_tweet" : "This is a fake tweet",
    "likes_count" : "10",
    "retweet_count" : "15",
    "tag" : "Elon",
},
{
    "profile_image_url" :
        "https://pbs.twimg.com/profile_images/1255961776035729408/0AjDFNaf_400x400.jpg",
    "author" : "Elon Musk",
    "handle" : "@elonmusk",
    "time" : "2 min",
    "text_of_tweet" : "This is a fake tweet #2",
    "likes_count" : "10",
    "retweet_count" : "15",
    "tag" : "Elon",
}]
```

Moya is amazing because it allows you have a ‘SampleData’ portion of the network request. This means that if you flip a switch on the moya networking layer, it will return a standardized data that you freezed from the server. This makes testing infinitely easier since you can just test to verify that the known data is still being processed correctly. I made a simple python script to make this process easier, generating sample data files in response to a variety of network requests. These files can then be placed into the ‘sampledata’ folder in the xcode project and directly referenced using “stubbedResponse(‘datafilename’)”. This makes the Unit Testing portion of the application much easier and streamlined.

Unit Testing

Overall Testing and Authentication




```swift
// MARK: ---------------------- Tests that require Network Connection: ----------------------

func loginToAdmin(_ networkProvider: NetworkManager) {
    let logoutExpectation = self.expectation(description: "Logging in with Cognito")
    congnitoLogout() {
        logoutExpectation.fulfill()
    }
    waitForExpectations(timeout: 5, handler: nil)

    let expectation = self.expectation(description: "Logging in with Cognito")
    cognitoLogin(networkProvider, username: "nealsoni00@gmail.com", password: "nealsoni") {
        print("[AWS] Congito logged in")
        expectation.fulfill()
    }
    waitForExpectations(timeout: 5, handler: nil)
}

// Need network connectivity to test this with .normal
func testAdminRealLoginTime() {
    let networkProvider = NetworkManager(networkAdapter: NetworkAdapter(providerType: .normal, testAccountType: .Admin))
    loginToAdmin(networkProvider)
    let expectation = self.expectation(description: "Getting refresh with server")
    networkProvider.networkAdapter.refreshToken(withProvider: networkProvider) { success in
        if success {
            expectation.fulfill()
        }
    }
    waitForExpectations(timeout: 5, handler: nil)

    self.measure {
        let expectation2 = self.expectation(description: "Get All App Data")
        networkProvider.getAppData() {
                expectation2.fulfill()
        }
        waitForExpectations(timeout: 5, handler: nil)
    }
}
```

Left: This is the unit testing framework we developed to allow for streamlined testing. These tests test all the properties of the network manager, using the stubbed data shown above returned and injected into the views by the Moya layer. This decouples the project from the server, ensuring that any test cases that fail fail because of the application, not because of failed network requests or because of the server.
Right: This is the login unit test for verifying that the user is logged into the server and has the appropriate cognito access token on his local device. It also runs through the login process to verify that the user gets the appropriate refresh token from the stubbed 'server'.

Simulating Network Requests

```
func getAnnouncements(_ networkProvider: NetworkManager) {
    let getAllAnnouncements = self.expectation(description: "Get AllAppDataExpectation")
    networkProvider.getAllAnnouncements(completion: { () in
        XCTAssertEqual(networkProvider.adminAnnouncements.count, 4)
        XCTAssertEqual(networkProvider.studentAnnouncements.count, 4)
        getAllAnnouncements.fulfill()
    })
    waitForExpectations(timeout: 10, handler: nil)
}

func testGetAnnouncements() {
    // Announcements depend on the User Data and AlertTypes
    getAdminRefreshToken(networkProvider)
    getAlertTypes(networkProvider)
    self.getAnnouncements(self.networkProvider)
}
```

This is the code for a single test: Get Announcements. This would get all the tweets from the faked 'server', which according to the stubbed file, should be 4. If at any step of the fetching, decoding, and parsing process there was an error, 4 announcements would not be returned and the test would fail. The next step of this process is to ensure that these 4 messages are displayed on the screen correctly.

Ensuring Views Process Data and Display Correctly

```
func testAnnouncementsPage() {
    getAdminRefreshToken(networkProvider)
    getAlertTypes(networkProvider)

    let announcementsStoryboard: UIStoryboard = UIStoryboard(name: "Announcements", bundle: nil)
    guard   let announcementsScene = announcementsStoryboard.instantiateInitialViewController() as? UINavigationController
            let view = announcementsScene.viewControllers.first as? AnnouncementsVC else {
                XCTFail("Can't instantiate view controller")
                return;
    }

    view.networkProvider = self.networkProvider
    // Load the view with no announcements fetched
    view.viewDidLoad()
    var expectedStudentAnnouncementCount = 0
    XCTAssertEqual(view.tableView.numberOfRows(inSection: 0), expectedStudentAnnouncementCount)

    // Refresh through the view's refresh method
    view.refresher.beginRefreshing()
    view.shouldRefreshAnnouncements(self)

    // Analyse the table view to see if it has the correct properties after the refresh
    XCTAssertEqual(view.tableView.numberOfSections, 1)
    expectedStudentAnnouncementCount = 4
    XCTAssertEqual(view.tableView.numberOfRows(inSection: 0), expectedStudentAnnouncementCount)

    for row in 0 ... view.tableView.numberOfRows(inSection: 0) - 1 {
        guard let currCell = view.tableView.cellForRow(at: IndexPath(row: row, section: 0)) as? AnnouncementCell else {
            XCTFail("Can't instantiate Announcements View Controller")
            return;
        }
        //Verify the cell data is processed correctly
        XCTAssertEqual(currCell.messageLabel.text, currCell.data?.message)
        XCTAssertEqual(currCell.titleLabel.text, currCell.data?.title)
    }
}
```

This is why Dependency injection is so important. We are able to inject a new network manager that we add whatever amount of data to that fake requests, and when we fake instantiate the view we can check to see that each of the theoretical cells are of the right type and have the right data in them. This is super useful for quickly verifying the data of hundreds of views without having to run the end-to-end User Interface tests described in the next section.

Notice the fact that we generate our own ‘networkProvider’ and inject it into the view. We then refresh the view with this new networkProvider. This ensures that the data is processed according to what we want it to display using our faked data, not the actual network provider connected to the server.

This test shows that when we turn through the returned stubbed data, the data on the announcementsVC is parsed correctly by the table view — 4 cells will be displayed and they will all have the correct order of ‘message’ and ‘title’ properties.

UI, End-to-End Testing

```
func login(_ app: XCUIApplication, username: String, password: String){
    // Make sure we start on the login screen
    _ = waitForViewToAppear(app, name: "LoginVC")
    // Go to cognito login
    app.buttons["cognito"].tap()

    let tablesQuery = app.tables
    // Select and type the username
    let usernameLabel = tablesQuery.staticTexts["User Name"]
    usernameLabel.tap()
    let usernameInput = tablesQuery.cells.containing(.staticText, identifier:"USER NAME").children(matching: .textField).elemen
    usernameInput.tap()
    usernameInput.typeText(username)
    // Select and type the password
    let passwordLabel = tablesQuery.staticTexts["Password"]
    passwordLabel.tap()
    let passwordInput = tablesQuery.secureTextFields.containing(.button, identifier:"Show").element
    passwordInput.tap()
    passwordInput.typeText(password)

    // Press the login button
    app.buttons["Sign In"].tap()
}

func testCognitoLogin() {
    let app = XCUIApplication()
    app.launchArguments.append("--uitesting")
    app.launch()
    self.login(app, username: "nealsoni00@gmail.com", password: "nealsoni")

    let reportVC = waitForViewToAppear(app, name: "ReportVC")
    let primaryButton = reportVC.children(matching: .other).element(boundBy: 0).children(matching: .other).element(boundBy: 1)
    primaryButton.press(forDuration: 2.4)

    _ = waitForViewToAppear(app, name: "CustomMessageVC")
    app.scrollViews.otherElements.buttons["Send Update"].tap()

    app.buttons["settings"].tap()
    app.tables["SettingsVC"].staticTexts["Logout"].tap()

      app.swipeLeft()
      app.swipeLeft()
      app.swipeLeft()
}
```

These tests are actually super cool since they simulate an action being done on the screen of the phone. Essentially it searches for a specific button, text input, and recreates the action that the user would do on the phone for you to make sure that process still works. It does this by actually installing a seperate app on the phone that overrides the phone's screen / DOM and searches for elements to click or input text into.

These tests are not as scalable as the Unit tests since they require an actual device to simulate the process, need to be built on a mac machine, and lastly take much longer to run since they require UI graphics simulation. Additionally, the simulator on iOS does not have the ability to receive notifications so we were unable to test the critical alerts using UI tests — which is a big gap in testability on a build server. We were able to test this on our local phones but in production that would be an issue.

Section 4: Android Project

Android was much more challenging to work on than iOS. This is because every android device is different, and they all run different versions or flavors of android, on top of there being a dozen or so releases still being used today by users across the world. It is a very fragmented market which makes the job of any QA team infinitely more difficult. We were unable to get to the testing portion of the Android code since there was so much difficulty in making the views as production grade as possible and handling the critical alerts.

Receive One-Signal Critical Alerts

We had many difficulties using one signal for this process since one important part was that the notification never fails to deliver. Since Android handles notifications much differently than iOS, for example there has to be a background service open at all times for the notification to be delivered and processed on the phone, using One Signal had a few issues. When the phone was asleep for more than 3-4 hours, the notifications would not deliver correctly, if at all. This would be disastrous if the user didn't receive the alert on time. We decided to switch to sending notifications using firebase since that background service built by google worked much better and persisted for longer periods of time.

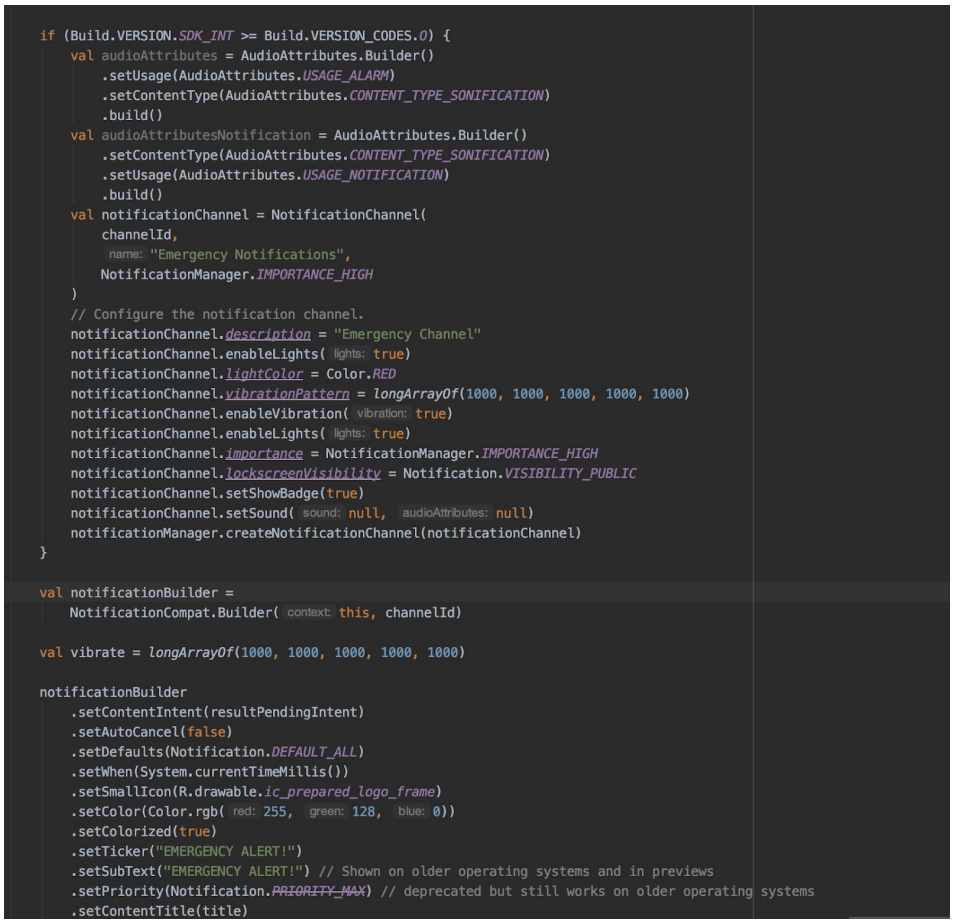
Notifications on Android are infinitely harder to understand and implement than on iOS for the sole reason that android has so many more different device types running different versions of android going back 6-8 years. Little of the devices in use today are on the most recent operating system. This made testing if notifications delivered super difficult.

There is no such thing as 'critical alerts' on android. Instead there is a setting that allows you to override the ringer by creating an alarm application. This is what we had to do to simulate the same style of delivery as the iOS application — except what took iOS 5-6 lines to accomplish, took android 150+ lines and more steps for the user. Essentially what happens is we display an instructions page describing what the user has to do when they press the 'begin' button on the bottom. Pressing that button displays a view where it lists every application and the permissions that application has. They have to scroll down till they see "TwitterAlerter" and then check the box under 'do not disturb permissions'. Once complete, they press the back button on the phone. This gives us permission to change the ringer and vibration settings — similar to iOS.

Firebase notifications are a challenge to deliver reliably and have code run in the service task to override the ringer, they have to be delivered as a 'data message' otherwise the 'one receive' code won't be run when the application is in background. Thus, when the application is in the

background, special precautions have to be taken to make sure the notification is seen by the user on every different OS level and flavor.

Notification Code Snippet



```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
    val audioAttributes = AudioAttributes.Builder()
        .setUsage(AudioAttributes.USAGE_ALARM)
        .setContentType(AudioAttributes.CONTENT_TYPE_SONIFICATION)
        .build()
    val audioAttributesNotification = AudioAttributes.Builder()
        .setContentType(AudioAttributes.CONTENT_TYPE_SONIFICATION)
        .setUsage(AudioAttributes.USAGE_NOTIFICATION)
        .build()
    val notificationChannel = NotificationChannel(
        channelId,
        name: "Emergency Notifications",
        NotificationManager.IMPORTANCE_HIGH
    )
    // Configure the notification channel.
    notificationChannel.description = "Emergency Channel"
    notificationChannel.enableLights( lights: true)
    notificationChannel.lightColor = Color.RED
    notificationChannel.vibrationPattern = longArrayOf(1000, 1000, 1000, 1000, 1000)
    notificationChannel.enableVibration( vibration: true)
    notificationChannel.enableLights( lights: true)
    notificationChannel.importance = NotificationManager.IMPORTANCE_HIGH
    notificationChannel.lockscreenVisibility = Notification.VISIBILITY_PUBLIC
    notificationChannel.setShowBadge(true)
    notificationChannel.setSound( sound: null,  audioAttributes: null)
    notificationManager.createNotificationChannel(notificationChannel)
}

val notificationBuilder =
    NotificationCompat.Builder( context: this, channelId)

val vibrate = longArrayOf(1000, 1000, 1000, 1000, 1000)

notificationBuilder
    .setContentIntent(resultPendingIntent)
    .setAutoCancel(false)
    .setDefaults(Notification.DEFAULT_ALL)
    .setWhen(System.currentTimeMillis())
    .setSmallIcon(R.drawable.ic_prepared_logo_frame)
    .setColor(Color.rgb( red: 255,  green: 128,  blue: 0))
    .setColorized(true)
    .setTicker("EMERGENCY ALERT!")
    .setSubText("EMERGENCY ALERT!") // Shown on older operating systems and in previews
    .setPriority(Notification.PRIORITY_MAX) // deprecated but still works on older operating systems
    .setContentTitle(title)
```

Notice on the top portion the code to determine if the application is running on a phone of Build Version O or higher (Android Oreo). This is the os version that notification channels were created which categorize messages into different groups and delivery importance. Also notice how the ‘proprity_max’ is crossed out on the .setPriority in the bottom. This is because that property has been deprecated in more recent versions of android but is still required to have the desired outcome on versions lower than O.

Network Manager

```
import com.serjltt.moshi.adapters.Wrapped
import okhttp3.ResponseBody
import retrofit2.Call
import retrofit2.http.*

interface TwitterAPI {

    @FormUrlEncoded
    @POST( value: "oauth/token")
    fun login(@Field( value: "client_id")   clientID: String,
              @Field( value: "assertion")   assertion: String,
              @Field( value: "provider")    provider: String,
              @Field( value: "grant_type")  grantType: String,
              @Field( value: "origin")      origin: String): Call<TokenData>

    @FormUrlEncoded
    @POST( value: "oauth/token")
    fun refreshToken(@Field( value: "client_id")      clientID: String,
                     @Field( value: "refresh_token")  refreshToken: String,
                     @Field( value: "grant_type")     grantType: String? = "refresh_token",
                     @Field( value: "origin")         origin: String? = "android"): Call<TokenData>

    @FormUrlEncoded
    @POST( value: "oauth/revoke")
    fun logout(@Field( value: "client_id")  clientID: String,
               @Field( value: "token")      token: String): Call<ResponseBody>
```

```
object NetworkManager {
    var sharedInstance : RestAPI = RestAPI()
    fun clearData(){
        this.sharedInstance = RestAPI()
    }
    // Called early on to initialize this class
    fun destroy(){
        this.sharedInstance = RestAPI()
    }
}

class RestAPI {
    var environment = Environment.Production
    // Make sure this is initialized in login and initial VCs
    // Initialized in onCreate of MainActivity
    var sharedPreferences: SharedPreferences? = null
    // Initialization Variables
    val preparedAPI: PreparedAPI
    val moshi: Moshi
    val retrofit: Retrofit
```

There were a lot of similarities between iOS and Android networking layers. The objects are compiled into json and the json is compiled back down into objects

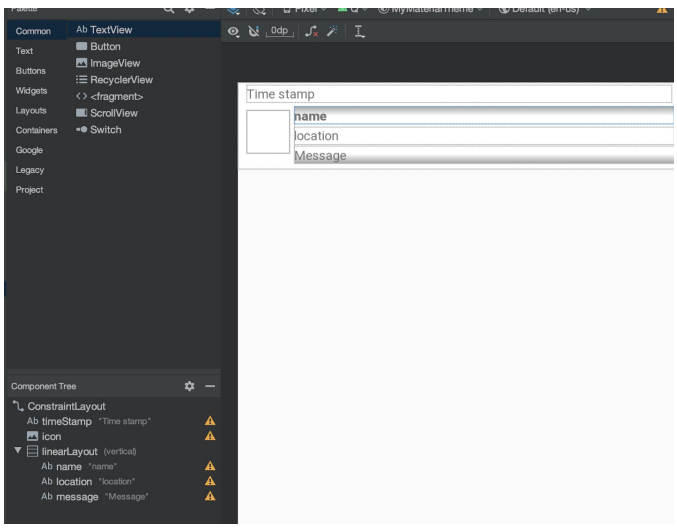
On Android I made the beginner's mistake of using a singleton reference to share the entire network manager with the whole project see on the right. We plan on using dependency injection, similar to how we migrated the entire iOS codebase away from the single-ton network object. This process will take a few weeks since the view code is tightly coupled to the idea that there is a network layer accessible at every level of the view's creation — described in the 'designing the interface' section later.

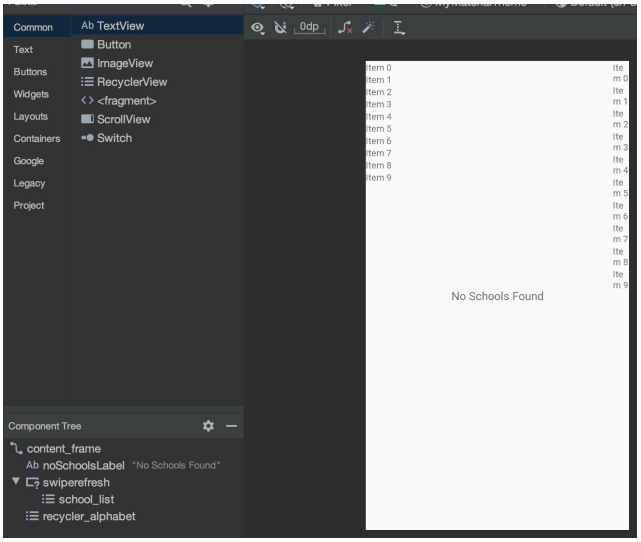
Data Model

```
@JsonClass(generateAdapter = true)
data class Announcement(@Json(name = "id" ) val id: Int,
                        @Json(name = "alert_tag_id") val alertTagId: Int?,
                        @Json(name = "alert_tag_name") val alertTagName: String = "",
                        @Json(name = "alert_type_id") val alertTypeId: Int? = null,
                        @Json(name = "cancelled" ) val canceled: Boolean? = null,
                        @Json(name = "did_initiate_response") val didInitiateResponse: Boolean? = null,
                        @Json(name = "emergency_level") val emergencyLevel: Int? = null,
                        @Json(name = "initiated_response_alert_type_id") val initiatedResponseAlertTypeId: Int? = null,
                        @Json(name = "is_emergency_report") val isEmergencyReport: Boolean? = null,
                        @Json(name = "location" ) val location: String? = "",
                        @Json(name = "message" ) val message: String = "",
                        @Json(name = "name" ) val name: String = "",
                        @Json(name = "school_ids") val schoolIds: List<Int>? = emptyList(),
                        @Json(name = "title" ) val title: String = "",
                        @Json(name = "delivery_type" ) val deliverTypeString: String? = "normal",
                        @Json(name = "user_id") val userId: Int? = null,
                        @Json(name = "created_at") val createdAt: String? = null,
                        @Json(name = "updated_at") val updatedAt: String? = null){
    val alertTag: AlertTag = AlertTag(alertTagName)
    fun alertType(): AlertType? { return NetworkManager.sharedInstance.alertTypes[alertTypeId] }
    val deliverType: DeliveryType = DeliveryTypes().fromKey(deliverTypeString)
    val createdDate: Date? = createdAt?.getDate()
    val updatedDate: Date? = updatedAt?.getDate()
    val createdAtFormated: String = updatedDate?.formatedString() ?: "Publication Time Unknown"
    val updatedAtFormated: String = updatedDate?.formatedString() ?: "Publication Time Unknown"

    override fun toString(): String {
        return "{$id, $title, $message, $name, $alertTag, ${alertType()}}"
    }
    //TODO: Convert the createdAt and updatedAt to date objects here
}
//Works
internal fun RestAPI.getAllAnnouncements(schoolId: Int? = null, success: ((List<Announcement>)->Unit)? = null, error: (()->Unit)? = null, failure

    val userType = user?.type

    val target = preparedAPI
    when (userType){
        UserTypes.Student, UserTypes.Teacher, UserTypes.Admin ->
            getAnnouncementsForSchool(schoolId,success,error,failure)
        UserTypes.DistrictAdmin, UserTypes.Police ->
            getDistrictAnnouncements(success,error,failure)
    }
}
```

This is almost identical to how the data models in iOS work, however, whenever network requests are made, they are sent with a data model which can be super confusing, especially when you wrap both the request objects and the response in another json key.

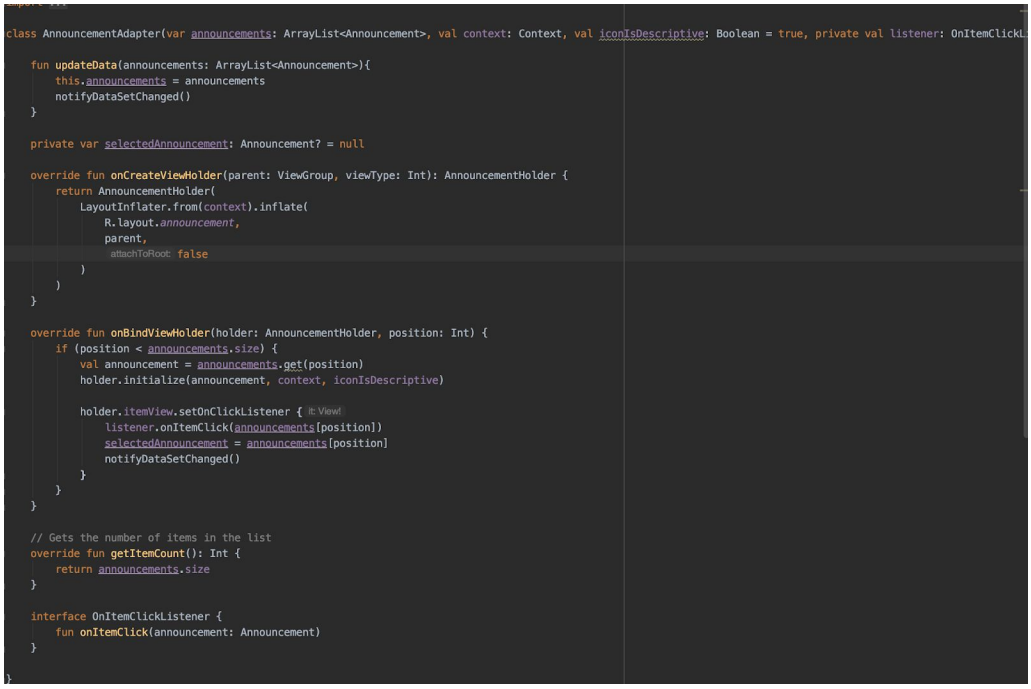
Designing the Interface





Interfaces are not nearly as enlightening or fun to design in Android. There is no storyboard, there is no way to see how the different objects interact, and there is no way to view the entire app hierarchy in one view. Above are the individual recycler view cell and on the right is a refreshable recycler view with date sectioned headers.

Displaying Recycler Views



```
class AnnouncementAdapter(var announcements: ArrayList<Announcement>, val context: Context, val iconIsDescriptive: Boolean = true, private val listener: OnItemClickL

    fun updateData(announcements: ArrayList<Announcement>){
        this.announcements = announcements
        notifyDataSetChanged()
    }

    private var selectedAnnouncement: Announcement? = null

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): AnnouncementHolder {
        return AnnouncementHolder(
            LayoutInflater.from(context).inflate(
                R.layout.announcement,
                parent,
                 attachToRoot: false
            )
        )
    }

    override fun onBindViewHolder(holder: AnnouncementHolder, position: Int) {
        if (position < announcements.size) {
            val announcement = announcements.get(position)
            holder.initialize(announcement, context, iconIsDescriptive)

            holder.itemView.setOnClickListener { it: View!
                listener.onItemClick(announcements[position])
                selectedAnnouncement = announcements[position]
                notifyDataSetChanged()
            }
        }
    }

    // Gets the number of items in the list
    override fun getItemCount(): Int {
        return announcements.size
    }

    interface OnItemClickListener {
        fun onItemClick(announcement: Announcement)
    }

}
```

The actual process of displaying these recycler cells is also extremely challenging. They require Adapters to connect announcement holders (similar to the cells from iOS) to the actual recycler views. Additionally, there are issues with bleeding of other messages into each other since these are ‘recycled’ cells. Each cell has to be cleaned before reuse, especially if properties are nullable between uses.

Section 5: Conclusion

Overall the whole independent study was extremely valuable to my learning of production application practices. I was really happy that there was so many online resources describing what steps to take, other people's most difficult challenges to overcome, and what bad-practices people had done in the past that they had to overcome. One example that related to was the idea of using singletons to share data between views on both Android and iOS. We originally made the network manager a singleton that was directly referenced inside each of the networkable views to get and refresh data with the server. This made the code extremely untestable since there was no way to decouple the view controller from referencing this singleton — and thus the singleton has to be instantiated before every test that was run and specific code had to be written inside the class to allow for it to be sorta tested. This became a large overhead, and after reading dozens of articles online, I found it was a programming mentality that many developers have had to overcome and it is actually very common. This was super cool to see that I was making the same mistakes people at actual companies were making, and I was solving them when they don't really matter — making a fake 'production ready' application instead of making the mistakes for a company or startup. This project was a good way to break the bad habits I had for both iOS and Android development.

It was also super cool to see the differences between writing Android Kotlin code and writing iOS Swift code, when trying to build the same application. The difference in designing the interface, the difference in making the networking layers, and how the same model-view-controller framework differed between the two was really interesting. I now understand why iOS and Android development is usually done by different people in a company but it is cool to know that I could do either going forward with relative ease.