Hinder

The Easy Way to Find Teammates

CS 130: Software Engineering
Professor: Miryung Kim

Discussion 1B
TA: Twinkle Gupta

Team Name: TBD
Kimberly Svatos, 604425426
Marshall Briggs, 304417630
Kyle Haacker, 904467146
George Archbold, 604407413
Daniel Berestov, 404441309
Apurva Panse, 504488023

Team Repository: https://github.com/marshdevs/Hinder

1 Design

1.1 UML Overview

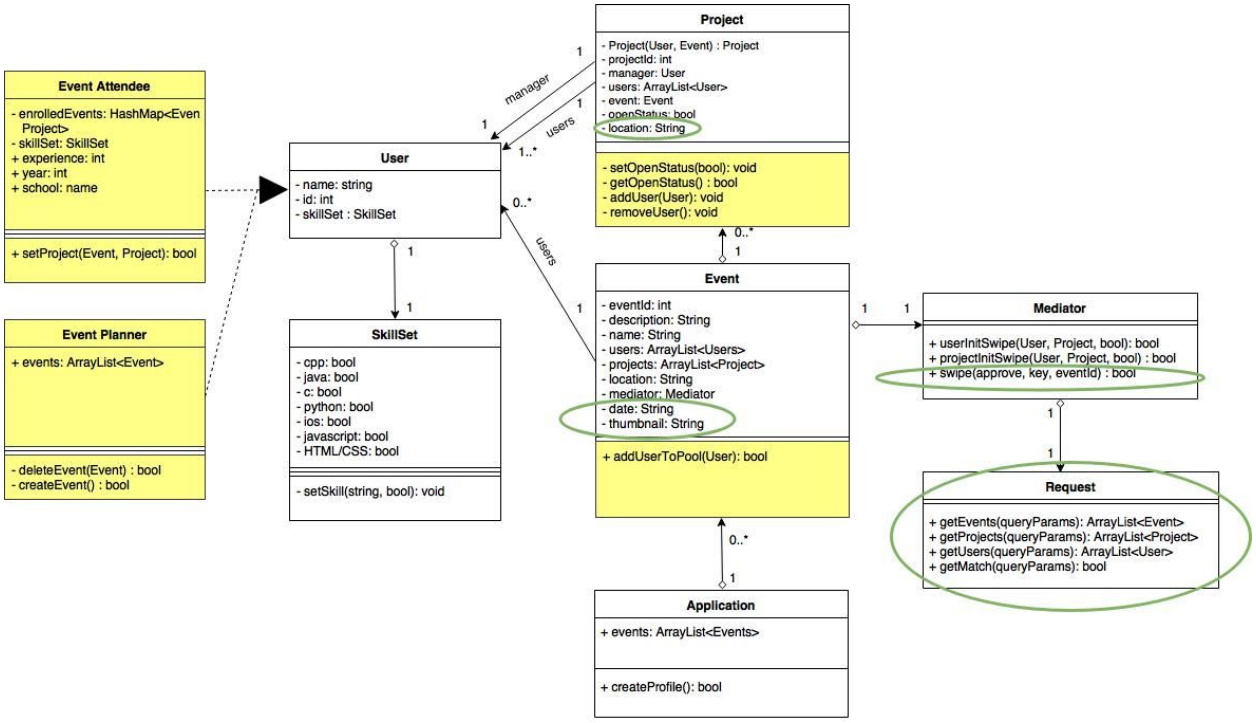


Figure 1: UML class diagram demonstrating changes since part A of the project. Classes and methods that are highlighted will be implemented for the final part C of the project. Values and methods that are circled have been added and implemented since the original UML diagram for part A.

Figure 1 shows the modifications and additions for our UML class diagram since part A. Accomplishments for part B include near-complete implementation of the core classes User, Project, and Event. Additionally, the swiping logic has been implemented in the Mediator class and a new Request class has been added to query the DynamoDB database. The next steps involve adding the specific Event Attendee and Event Planner roles and their associated functionality to the app. Users will soon be able to seamlessly update their profile information and manage their events and projects in the app. Additionally, part C will entail development of a very fun and intuitive swiping experience for users to find projects and vice versa.

1.2 Additional Libraries / Packages

Our project uses the **IGListKit** framework for creating UICollectionViews, such as the list of event data that is presented to the user on the app home screen. We also use **YogaKit** to simplify handling the layout of our views. Finally, we have decided that login information and profile pictures will be tied to Facebook accounts and hence have now included the **Facebook SDK** for our application.

1.3 iOS

1.3.1 Views

The HinderNavigationBar View class extends the Apple UIKit UINavigationBar class. It initializes all the relevant fields for our app navigation bar. The class includes an instance of the Apple UIKit UILabel class called titleLabel which is configured with backgroundColor, text, font, textAlignment, and textColor to give the header the desired look and feel. The HinderNavigationBar is initialized to add the titleLabel subview and set the bar color. We override layoutSubviews to include customized autoresizing, constraint-based behavior and set the title frame width.

The FeedEventCell and the FeedEventCellWithHeader Views are the view cells that live below the HinderNavigationBar. Each view represents a single event relevant to the user. The top cell is a FeedEventCellWithHeader and all subsequent cells are FeedEventCell views. This creates a list of all events that a user might be interested in joining. These views both extend the Apple UIKit UICollectionViewCell class and input the respective data information to provide the user with the event's specifics. The event data information is represented as a subview to the contentView property and added upon initialization.

The HinderCreateProfile View is responsible for handling the Facebook user authentication which is the main driver and information source that we use to create a user profile. Facebook authentication proved to be the best method for our app because Facebook provides us with all the relevant information we would need (name, age, contact info, profile picture) and there already exists an API to connect to Facebook. The HinderCreateProfile View class extends the Apple UIKit UIViewController, UIImagePickerControllerDelegate, UINavigationControllerDelegate classes. The class has a profilePic object which extends the Apple UIKit UIButton class and provides the

frame size information for user profile pictures. Upon loading the view the profilePic object gets a target action added to is that links it to Facebook and it is added as a subview. We defined a function to handle the Facebook user login authentication and profile creation. We initiate a Facebook login call and then if the passed permissions return success, we proceed to make a Facebook SDK request and pull all the relevant information we need to populate the newly created user account.

1.3.2 ViewControllers

The HomeViewController manages the logic for the home page and flow to all subsequent views. It loads all the subviews which include a settings button that reroutes to the settings page, a hamburger menu button that opens the hamburger menu, and a listAdapter object containing all the event cells. It contains the outline to implement a location based search that populates the listAdapter object with Events close to the User.

The EventPageViewController manages the logic for the display of an event view page. This view contains all the actions that you can do on one event page and is loaded when a user wants to see more information about a certain event.

1.3.3 Models

Event.swift extends NSObject and serves as the class that represents an event. It contains all the important fields for an event object and also an init function that parses a json file to extract all the relevant data fields.

Project.swift and User.swift are both similar to Event.swift however Project includes one key function, addUserToProject(userId: String). This is the function that takes care of project formation once the logic for swiping and matching signifies that a pair has been found. Skillset.swift will outline a template of predefined skills that will exist inside a user object as a key way for projects and users to identify if they are good candidates for a match.

Mediator.swift acts as the logic class that handles all the swiping. It is named after the design pattern because it acts as the central point of contact for all swipes between users and projects. There is one main swipe function that executes the server query, but there are two separate functions to deal with different logic for a swipe initiated by the user or by a project.

Request.swift model file encapsulates all HTTP request function logic that enables the app to communicate the server and make database queries. It has one query function for each type of object that the app would need to get from or send to the database.

1.4 Node Server

1.4.1 App.js

All of the server code exists in the app.js file. There is a server endpoint defined for every possible database action. The main actions, being a query, where the app request information from the database, a creation, where the app wants to store a newly created object in the database, a deletion, where the app wants to remove an existing instance from the database, or an update, where certain fields of an existing database object are modified.

1.4.2 Request.json

The request.json file specifies information vital for interacting with our backend server:

(1) **How to receive data from the server:** Requests.json outlines the schemas for the four persistent data types (Event, Project, User, and Match). These schemas reveal how the server will return each data type.

(2) **How to send data to the server:** This file also dictates the json structure for all possible server calls. There are four main json objects used to communicate to the server - Events, Users, Projects, and Matches. It also includes in the comments the corresponding server endpoints for each object's PUT, POST, and DELETE call. These json objects include fields like ID, name, description and then also fields specific to that object.

1.4.3 Package.json

The package.json file contains all the necessary information to power our node server, and to keep documentation of creation information and project logistics. It includes a short description of it use, url for the homepage, git repository, and bugs, a test command to run the test script, and all of the app dependencies that we are using to power the backend of our project.

1.5 Dynamo

We will use AWS DynamoDB, a fast and flexible NoSQL database service, to implement persistency in our application. Our database's secret access keys and read/write permissions are delegated to our backend server, implemented in 'app.js', which serves as the sole gateway between our application and our database. This provides extra (vital) security measures, at the cost of some performance. In the future, we plan to make use of another AWS service, DynamoDB Accelerator (DAX: an in-memory cache for DynamoDB), to mitigate the decrease in performance.

Within our database, we have created four tables: 'hinder-events', 'hinder-projects', 'hinder-users', and 'hinder-matches'. In each of the tables, values are stored as JSON objects.The different types of interactions with the various tables (GET, POST, PUT, DELETE) are defined in app.js and implemented using the aws SDK.

- **'hinder-events':** Stores instances of the Event model as JSON objects. Indexed by eventId.
- **'hinder-projects':** Stores instances of the Project model as JSON objects. Indexed by projectId.
- '**hinder-users':** Stores instances of the User model as JSON objects. Indexed by userId.
- **'hinder-matches':** Stores swipe interactions between Hinder users and projects as JSON objects. Indexed by matchId, which is a string containing "<userId>&<projectId>".
  Schema: {matchId: String, approve: Boolean (What kind of swipe interaction already exists between these two?)}

1.6 User Interface

Our user interface currently consists of two views. There is a home screen that the user is presented with upon launching the app. The home screen lists events near the user. The user can also select the profile settings (top left) to access their profile information, or the menu (top right) to access and manage their events and projects. Aside from the current lack of available event data and some purely visual differences, the home screen is very similar to the mockup we presented in part A.

There is also currently a basic profile settings view that allows the user to view and update their profile information. This view is currently missing the profile picture component; we anticipate this being added in part C through use of the Facebook SDK.

Both views are presented below. With the bulk of the Request and Mediator logic now implemented, we fully anticipate that we will be able to develop and deliver the remainder of the UI, as presented in the part A mockups, by the part C deadline.

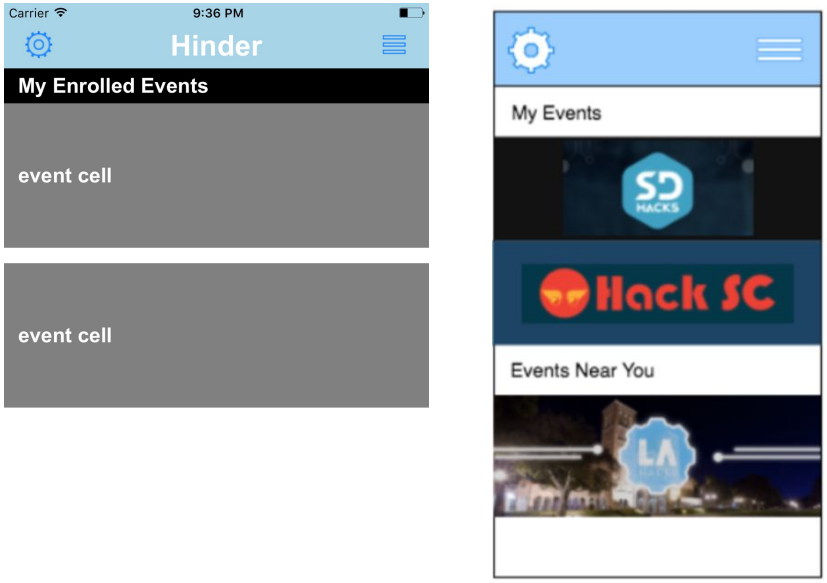


Figure 2: Current home screen of app (left) vs. mock-up presented in part A (right). Aside from the current lack of event data and some visual differences, the two views are very similar in design. A goal for part C will be to add a section for “My Events”.

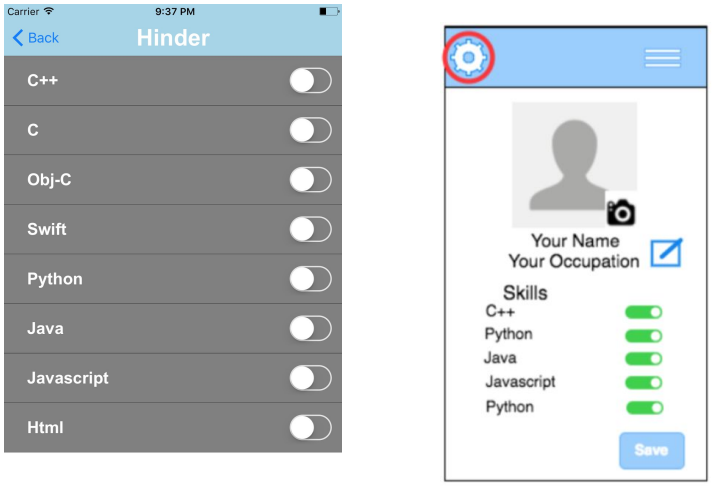


Figure 3: Current profile settings screen of app (left, work in progress) vs. mock-up.

2 API Description

The majority of our API functions currently live in Mediator.swift and Request.swift as these two modules are the main executors for the app logic. We implemented the XCode Markup formatting style in the Apple developer docs to document our code, following this example.

```
/**
 A swipe action.

 This function takes in swipe parameters
 and queries the database to see if a pair
 of matching User/Project IDs exists.

 - parameter approve: Left (false) or right (true) swipe
 - parameter key: The ID to search the database
 - parameter eventId: The corresponding ID of a previous swipe

 - returns: A true or false indicating existence of the match
 */
func swipe(approve: Bool, key: String, eventId: String) -> Bool {
    let direction = approve ? "/right/" : "/left/"
    return Request.getMatch(params: "match/" + eventId + direction + key)
}
```

Quick Help

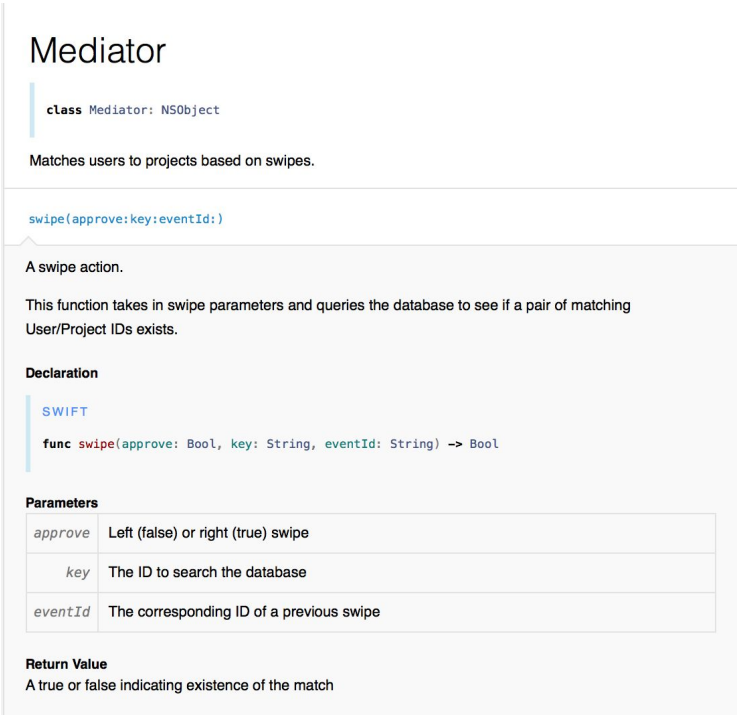
Declaration	`func swipe(approve: Bool, key: String, eventId: String) -> Bool`
Description	A swipe action. This function takes in swipe parameters and queries the database to see if a pair of matching User/Project IDs exists.

Parameters	`approve`
	Left (false) or right (true) swipe
	`key`
	The ID to search the database
	`eventId`
	The corresponding ID of a previous swipe

Returns	A true or false indicating existence of the match
Declared In	Mediator.swift

Full details for each API in Mediator.swift and Request.swift - including argument types, names, and return types - can be viewed in XCode via Quick Help. Alternatively, these details can be viewed in the webpages generated by Jazzy (similar to Javadocs,

converts markup in XCode to HTML). The API webpages can be found under directory APIDocs in the GitHub repository. See the example API description below.



Mediator

```
class Mediator: NSObject
```

Matches users to projects based on swipes.

`swipe(approve:key:eventId:)`

A swipe action.

This function takes in swipe parameters and queries the database to see if a pair of matching User/Project IDs exists.

Declaration

```
SWIFT
func swipe(approve: Bool, key: String, eventId: String) -> Bool
```

Parameters

approve	Left (false) or right (true) swipe
key	The ID to search the database
eventId	The corresponding ID of a previous swipe

Return Value
A true or false indicating existence of the match

Thus far, we are confident that our API design is adhering to the information hiding principle. Consider the Mediator class first. The notion of swiping on groups and/or users of interest is arguably *the* key feature of the app, and we don't expect the main swipe functionality to change over the evolution of the app. Details aside, a swipe will always involve a user, a project, and decision (was it a left swipe or a right swipe?) and hence we feel confident including these design decisions in the interfaces. A design decision that *may* be likely to change for our Mediator class is the exact details of the swipe logic. Depending on the evolution of the app, additional logic (e.g., possibly based on location or time) may be added for when users and projects swipe on each other. This design decision is hidden within the userInitSwipe and projectInitSwipe APIs. This allows for a great deal of flexibility in deciding what to implement or add for swipe logic as the app evolves.

The argument can also be made that the Request class is adhering to the information hiding principle. Namely, users of the Request APIs need only provide the search query without needing to know the underlying layout or storage of the data in the backend. One feasible evolution scenario is that additional event-related data is later added to the database. The Request class would be modified to retrieve this additional information

and encapsulate it in the Event object that is returned. The client will still receive all information in the Event object, without needing to know or account for the fact that the underlying storage may have changed. A similar argument holds for the getProjects (e.g., additional project-related data is added later) and getUsers (e.g., additional user profile data is added later) APIs provided by Request as well.

The change matrix below gives some estimated evolution scenarios and the expected changes needed in our key modules for each scenario:

	Application View	**Mediator**	**Request**
Filter potential swipe choices based on location or time		**X**	
Add additional user profile information (e.g., school)	**X**		**X**
Add additional event information (e.g., see event coordinator, companies attending event)	**X**		**X**
Add smooth and intuitive swiping gestures	**X**		
Limit the number of times users and projects can re-swipe on one another		**X**	
Allow users to upload resumes and/or transcripts	**X**		**X**

Finally, we present sequence diagrams to capture how the app will be using some of the important APIs that we have developed so far. As a reminder, full API documentation for part B can be found in the GitHub repository, in the source files or by looking through the APIDocs directory.

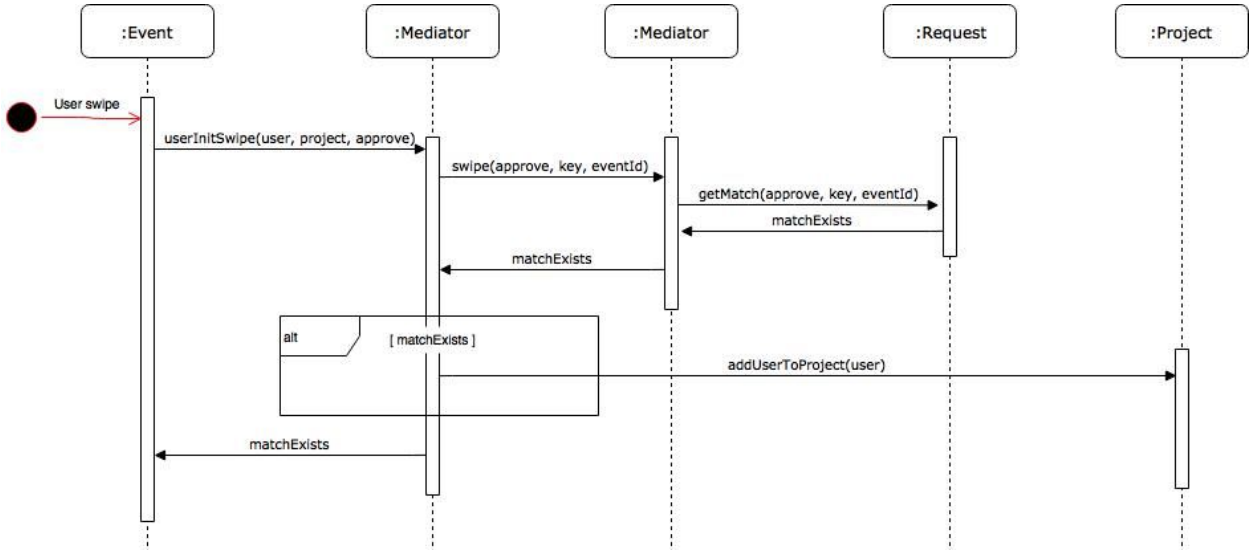


Figure 1: Sequence diagram demonstrating API use for swipe behavior. User swipes are first handled the Mediator APIs userInitSwipe and swipe. Mediator uses the Request's getMatch API, which queries the database, to determine whether the user has matched to the project. If a match exists, the user is added to the project. The logical flow for a project swiping on a user is very similar to the flow shown above.

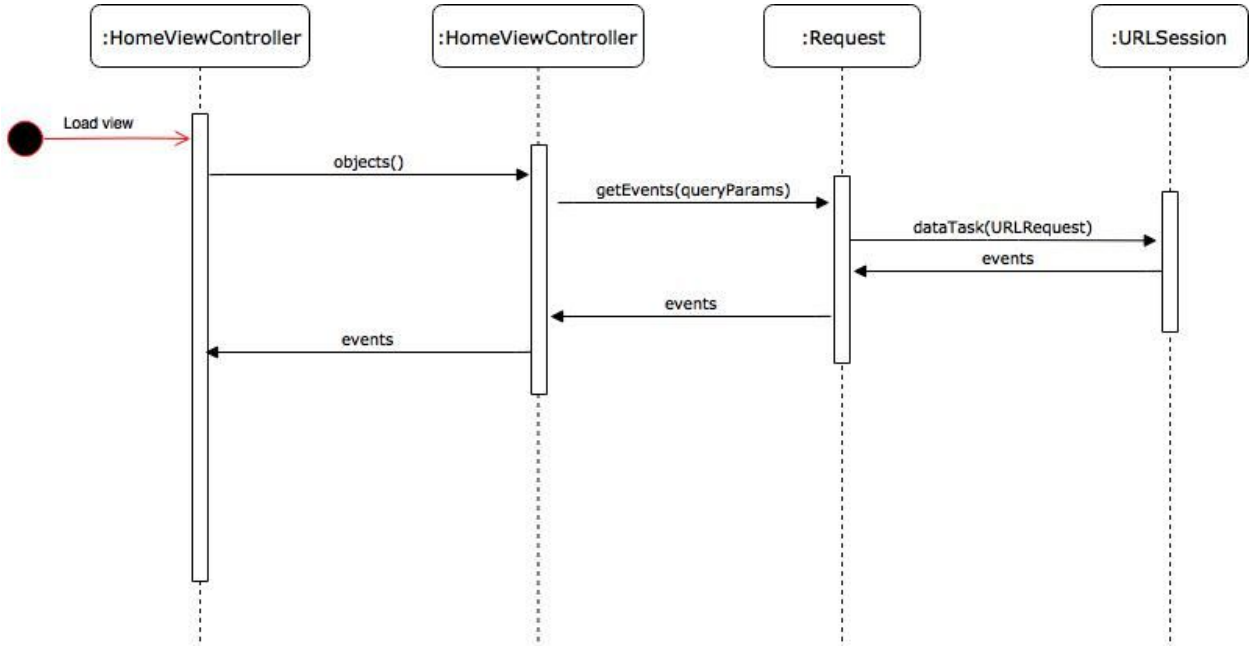


Figure 2: Sequence diagram demonstrating API use for retrieving event data. The logical flow for getting project data (API: getProjects) or user data (API: getUsers) is very similar. Information hiding is achieved in that the view controller is able to query for event, project, and user data without needing to know exactly how to retrieve the data.

3 Testing

As we have been creating this application, we have had to write test cases to make sure our View Controllers and files are functioning correctly. In the ViewControllerTests.swift we have written the test cases for our view controllers. There have not been many conditional statements to write and for view controllers there are few paths that can be taken, so the majority of the tests are spent checking that the objects in the View Controllers are being created correctly and are being updated when functions are called correctly.

Specifically HomeViewController has a UICollectionView which has different cells depending whether the cell is the first one of its kind or not. We make sure to check that the correct cell is being inserted into the UICollectionView in our test for the HomeViewController. We will perform similar tests when later ListSectionControllers are written for our application.

```
if(controller.collectionView.cellForItem(at: IndexPath(row: 0, section: 0)) != nil) {
    XCTAssertTrue(((controller.collectionView.cellForItem(at: IndexPath(row: 0, section: 0)) as?
        FeedEventCellWithHeader) != nil))
}
```

Above is an example of testing of the collectionView. If the cell at that index and path is nil which means we have added no events we won't enter this test, but we have two entries in DynamoDB currently so two cells are added to HomeViewController's UICollectionView. In this test we make sure that the first cell is of type FeedEventCellWithHeader which is the implementation we want. If that is not the case our assert statement will fail and we will know there is a bug in our code.

In our HinderCreateProfile View Controller we have to access the Facebook API and upload the profile picture to AWS S3 so this test cannot be automated fully. It will require a user to cancel the Facbook log in to test one path and then to succeed with the log in triggering another path. Once the user log ins to their Facebook account their profile picture is sent to our s3 bucket where it is stored with their unique user id. To test this view controller in the future we will redownload the image from s3 and compare the picture that we uploaded to the one we then downloaded from the s3 bucket and if their internal data is identical our service works correctly. This similar testing process will be done for all of our event and projects as they upload pictures to their corresponding s3 buckets.

Once we test the View Controllers we also have to test our Mediator and Request swift files. Unfortunately we were not able to write the tests for this part of the project since we haven't completed full implementation for testing but we know what we want to test. In Request we have the functions getEvents, getProjects, getUsers and getMatch. Based on a query getEvents, getProjects, and getUsers return an array of Events, Projects, and Users respectively. Testing would involve creating a mixture of Events, Projects, and Users and uploading them to DynamoDB. Then when we pull these objects from DynamoDB we will check that not only are the correct objects being pulled based on the query, but that the object is equivalent pre and post storage in DynamoDB. We will do this by testing the equality of the internal data structures of these objects. The getMatch function is slightly different because it returns a boolean value indicative if an entry exists in the database signifying that a user has right swiped on a project or vice versa. To test this function we would simulate a user swiping on a project and vice versa (this is part of the Mediator class testing) and check that an entry exists in the database for this action and then we would also make sure that no entry exists for a project and user if they have not swiped on each other. As mentioned previously the Mediator is responsible for handling swipes. The swipe function calls getMatches so to test this function we would call it with a left and right swipe and make sure our logic works correctly for both paths. For the userInitSwipe and projectInitSwipe functions the two paths taken for both functions are the user is added to the project if there is a match and otherwise returns false telling us there has been no swipe on a party by either of the user or the project.

Ultimately, we have performed as much testing of our files as possible and we understand that there is still much testing to be done for the Mediator and Request classes along with other Model files. We have used the iOS Unit Testing Bundle, however we also plan on incorporating the iOS UI Testing Bundle once our application starts having more cohesion between views.