Signfi – Cloud computing final project

Shenxiu Wu, Tingran Yang, Yang Hu, Zihao Zhou

Abstract

GitHub Link: $https : //github.com/bifeitang/Signfi_C C_f inal.git$
Youtube Link: $https : //youtu.be/ghYjgXT75Nc$

1 Introduction

Signfi, named after a combination of sign in and selfie, is an ios APP we developed to recognize all the people in one or several live group photos and check them as presence. This APP helps a lot with teachers who are bothered by the attendance of the students. Now the teachers just need to upload all the students selfie beforehand, and take a group photo with students at anytime to sign all the students in. Tour guides, who want to always be aware of whether all the guests are still stick with the team, may also benefit from this APP. In both cases, Signfi provides a friendly and convenient option.

In Section 2, we will introduce Amazon Web Services(AWS) components we used in this project and how these components are organized to realize the Signfi facial recognition system. In section 3, we will walk through the code and discuss in detail how each building block works. Then we will show our result in Section 4, and our future work in Section 5 [1].

2 Key Components

We base our APP on the AWS Cloud which offers S3 storage, Lambda function, Cognito user authentication, AWS Rekognition and DynamoDB NoSQL database. Besides, AWS also proposes a convenient way to develop IOS APP with Mobile Hub, who integrates S3, Cognito, DynamoDB, etc as SDK. One can easily introduce this SDK in the frontend to interact with backend. For the image recognition, Amazon Rekognition provides APIs to do the facial recognition with a high accuracy.

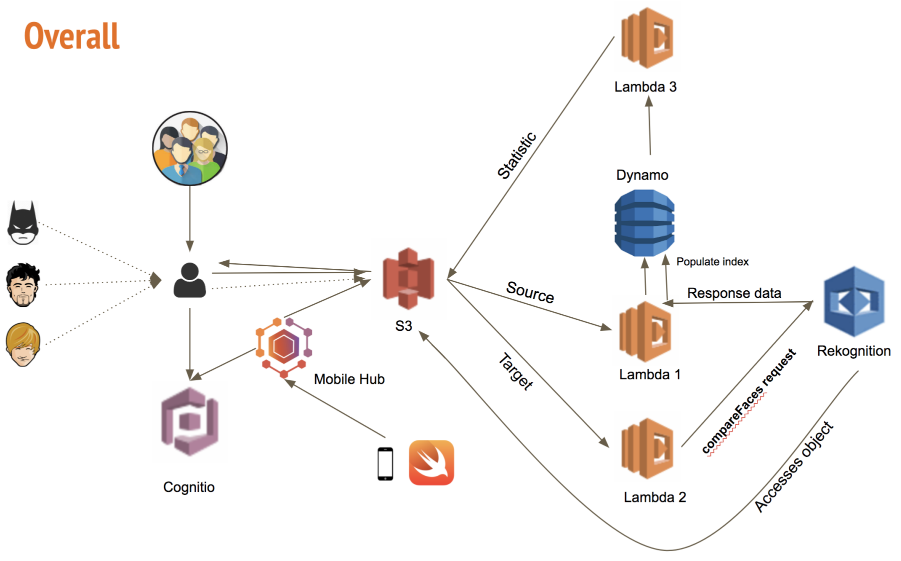


Figure 1: Overall Structure

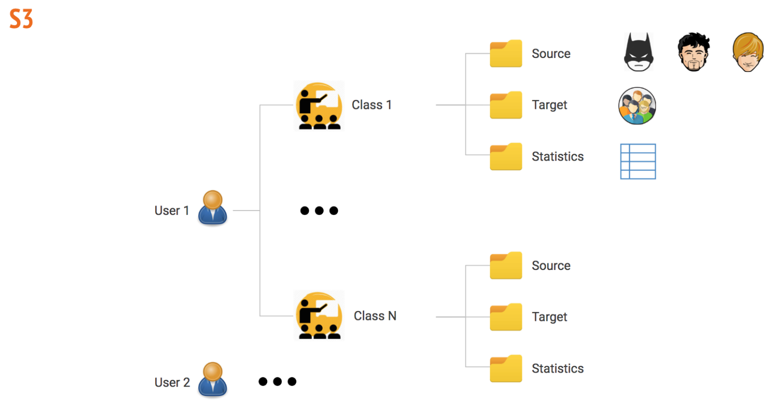


Figure 2: Overall S3

As illustrated in Figure 1, users can first sign in from their ios device, which is managed by AWS Cognito. Then upload an image from their ios device S3 bucket. Notably, this service is pre-configured by AWS Mobile Hub and we are using its nice isolation feature in our project, where each signed in user only can upload and read image from their corresponding folder. As shown in Figure 2, each user can create multiple folders in their own folder. Every folder contains three folders, which are Source, Target and Statistic. The Source folder has students personal info uploaded, like each student's image and named with the student's name or UNI. The Target folder contains all the group pictures uploaded by the user. And the statistic folder stores the analyzing results from back end.

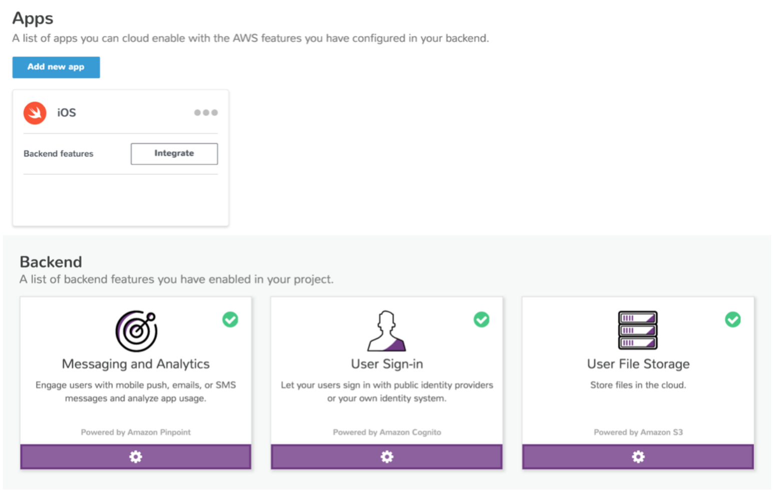
At present, pictures in source folder are manually uploaded by us directly from back end, but we will integrate this function to the front end soon. Suppose it functions, this APP works in 4 Steps.

- Create Class/Group: once user tab create new group in the front end, we will create a new class folder in S3 bucket under that user's folder with Source, Target and Statistic folder contained. The user can modify the name of folder or delete from front end.
- Create **Source Set**: a set of people app users want to recognize in each class or group. User can create this Source Set by uploading pictures to the source folder in S3 and specifying the pictures' name as the people's name. Each Source Set correlates with a table in DynamoDB, which contains two columns – user name and attendance status. Whenever the files in source folder are modified, **Lambda 1** will be triggered to change the table in DynamoDB correspondingly.
- Recognize Selfie Image: upload a group photo to Target folder and trigger the **Lambda 2** function which is connecting to Amazon Rekognition, as shown in Figure 1. Inside the Rekognition, source images will be compared with Target image one by one and the result will be wrote into DynamoDB corresponding table.
- Do Statistics: changes of the DynamoDB will trigger the **Lambda 3** which will create a csv file recording this newly-come Rekognition analysis, send this file to the statistic folder inside S3 and update the "history.csv" in the statistic folder.

When all these steps are finished, the user can easily check the presence of people in each group easily from the APP. And even their historical experience. We may also provide the download of these statistical results later in our IOS App.

3 Implementations (Code)

In this part, we will walk through the implementation of our app in detail. Section 3.1 talks about how we integrate the Mobile Hub to the swift code in IOS development, including the interaction with S3 bucket and Cognito. Section 3.2 introduces those three lambda functions.



A list of apps you can cloud enable with the AWS features you have configured in your backend.

A list of backend features you have enabled in your project.

Engage users with mobile push, emails, or SMS messages and analyze app usage.

Let your users sign in with public identity providers or your own identity system.

Store files in the cloud.

Figure 3: Overall S3

3.1 Mobile Hub (Front End)

As shown in Figure 3, Mobile Hub provides easy integrates of AWS services of back end. In our project, the Cognito (User Sign-in in Figure 3) and User File S3 (User File Storage) are used. This will give a app specific cloud configuration file, which I can add to my IOS app info.plist.

Another basic set up is the AWS SDK. A very basic and easy way, which is also used in our project is installing with Cocoapods. One can init the CocoaPods project by running the pod install command from the folder containing the projects *.xcodeproj file. The explanation down below will assume that we can successfully import all the AWS packages needed.

3.1.1 Interact with S3 Bucket

There are two major information for uploading the image, one is the key and the other is the data. Taking the upload of target image as an example. Once the upload button is tapped, the askForImagename function will be triggered as in line 3. Since the prefix is already configured to the group-name/target/, we can ask the user to input a image name for the picture to be uploaded and append the name to the prefix as unique key we are going to use to upload image (at line 15). The data is the image present at the image interface, which can be get with askForImagename function at line 3. With the data and key info, the manager will group them into local content (in line 24). The manager here is AWSUserFileManager.defaultUserFileManager(). Which is a AWSUserFileManager imported from AWSMobileHubContentManager. Then with function uploadWithPin, one can uploads data associated with the local content at line 29. During the uploading process, we can update the progress bar inside the progressBlock and hand the failer in the completionHandler function.

```
1   @IBAction func uploadButtonTabbed(_ sender: UIButton) {
2           guard let image = sefiImageView.image else {return}
3           self.askForImagename(UIImagePNGRepresentation(image)!)
4   }
5
6   prefix = nextcontent?.key as Any as! String + "target/"
7   fileprivate func askForImagename(_ data: Data) {
8           ......
9           let doneAction = UIAlertAction(title: "Done", style: .default) {[unowned self] (action:
             ↪  UIAlertAction) in
10              let specifiedKey = alertController.textFields!.first!.text!
11              if specifiedKey.characters.count == 0 {
12                  self.showSimpleAlertWithTitle("Error", message: "The file name cannot be empty.",
                     ↪  cancelButtonTitle: "OK")
```

```
                return
            } else {
                let key: String = "\(self.prefix!)\(specifiedKey)"
                self.uploadWithData(data, forKey: key)
            }
        }
        ......
}

// Upload the files
    fileprivate func uploadWithData(_ data: Data, forKey key: String) {
        let localContent = manager.localContent(with: data, key: key)
        uploadLocalContent(localContent)
    }

    fileprivate func uploadLocalContent(_ localContent: AWSLocalContent) {
        localContent.uploadWithPin(onCompletion: false, progressBlock: {[weak self] (content:
         ↪  AWSLocalContent, progress: Progress) in
                guard self != nil else { return }
                self?.progressView.progress = Float(progress.fractionCompleted)
            }, completionHandler: {[weak self] (content: AWSLocalContent?, error: Error?) in
                guard let strongSelf = self else { return }
                if let error = error {
                    print("Failed to upload an object. \(error)")
                    strongSelf.showSimpleAlertWithTitle("Error", message: "Failed to upload an
                     ↪  object.", cancelButtonTitle: "OK")
                }
        })
    }
```

3.1.2 User authentication with Cognito

To use the Cognito and S3 feature, we have to integrate the Amazon Pinpoint first, which is used to targeted push notifications to drive mobile engagement. Like capturing the user's uploading action we mentioned above. For the Cognito, we can just use the function AWSSignInManager.sharedInstance() which contains function like isLoggedIn(), logout(), login() to do basic log in.

```
var pinpoint: AWSPinpoint?
//Used for checking whether Push Notification is enabled in Amazon Pinpoint
static let remoteNotificationKey = "RemoteNotification"
func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions:
 ↪  [UIApplicationLaunchOptionsKey: Any]?) -> Bool {
    // Override point for customization after application launch.
    // Register the sign in provider instances with their unique identifier
 AWSSignInManager.sharedInstance().register(signInProvider:
  ↪  AWSCognitoUserPoolsSignInProvider.sharedInstance())
    let didFinishLaunching = AWSSignInManager.sharedInstance().interceptApplication(application,
     ↪  didFinishLaunchingWithOptions: launchOptions)

    pinpoint =
     ↪  AWSPinpoint(configuration:AWSPinpointConfiguration.defaultPinpointConfiguration(launchOptions:
     ↪  launchOptions))
    if (!isInitialized) {
            AWSSignInManager.sharedInstance().resumeSession(completionHandler: { (result: Any?,
             ↪  error: Error?) in
            print("Result: \(String(describing: result)) \n Error:\(String(describing: error))")
            })
        isInitialized = true
    }
    return didFinishLaunching
}
```

3.2 Rekognition and Lambda Function (Backend)

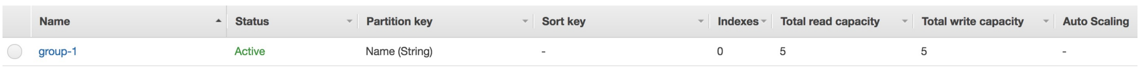
In the backend, the lambda function plays a key role as a connection, which bridges different components in AWS. The Lambda function is mainly composed of 3 stages. [2]

3.2.1 Stage 1

In stage 1, Lambda is triggered by S3 events when users upload source pictures into their private source folders under their user named folders in S3 bucket, this step is to preserve images of each person's face portrait beforehand. Once triggered, the lambda function calls AWS DynamoDB **CreateTable** API. **CreateTable** is an asynchronous operation. Upon receiving a **CreateTable** request, DynamoDB immediately returns a response with a TableStatus of CREATING. After the table is created, DynamoDB sets the TableStatus to ACTIVE. The created table in DynamoDB is named after user's name and the group name under the user's folder (Figure.3). In Figure.4, a primary key is created named "Name" with empty entries.

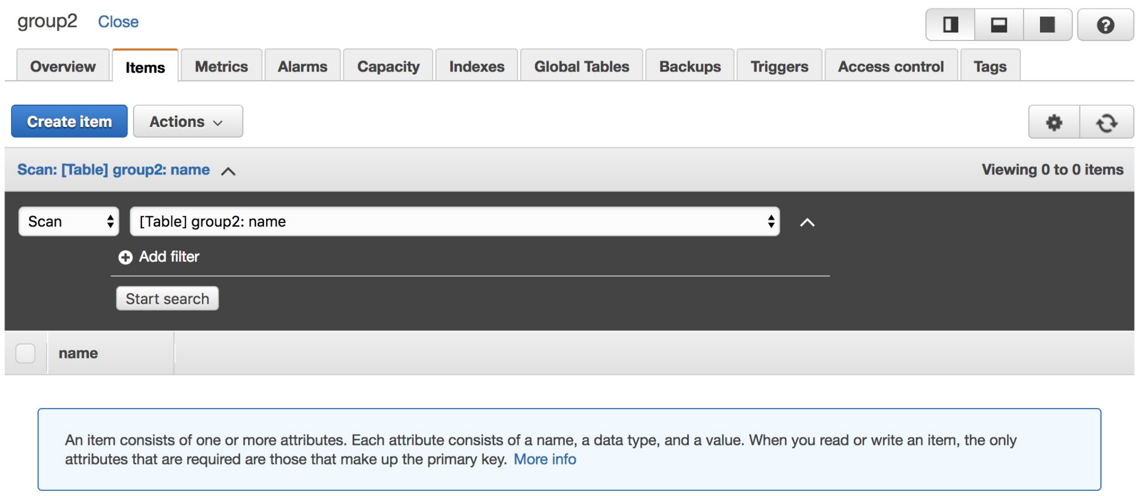
```python
def new_table(tablename):
    table = dynamodb.create_table(
    TableName=tablename,
    KeySchema=[
        {
            'AttributeName': 'Name',
            'KeyType': 'HASH'
        },
    ],
    AttributeDefinitions=[
        {
            'AttributeName': 'Name',
            'AttributeType': 'S'
        },

    ],
    ProvisionedThroughput={
        'ReadCapacityUnits': 5,
        'WriteCapacityUnits': 5
    }
)
```



	Name	Status	Partition key	Sort key	Indexes	Total read capacity	Total write capacity	Auto Scaling
○	group-1	Active	Name (String)	-	0	5	5	-

Figure 4: A table is created in dynamoDB



An item consists of one or more attributes. Each attribute consists of a name, a data type, and a value. When you read or write an item, the only attributes that are required are those that make up the primary key. More info

Figure 5: Empty entries in table

3.2.2 Stage 2

In stage 2, Lambda is triggered by S3 events when users upload target pictures (their group photos) into their private target folders in S3 bucket. For example, when the professor uploads one or several group photos of the whole class to record the attendance of his or her each student, the lambda will be triggered to call the AWS Rekognition with **compareFaces** request. Before sending out the request, lambda will also call **List Objects** API of S3 to get a list of source photos stored in the S3 private source folder. Then by using a FOR loop, lambda could iteratively get a single source photo return which is stored in the source folder before, namely in each one FOR loop, lambda will send a **compareFaces** request to call Rekognition with two parameters, which respectively are the path location in S3 bucket of each source photo and target photo. In this way, every source photo will separately be compared with the target photo, namely Rekognition will repeatedly make a judgment to determine whether the people in the source photo have appeared in the target photo. In response, the operation returns an array of face matches ordered by similarity score in descending order. For each face match, the response provides a bounding box of the face, facial landmarks, pose details (pitch, role, and yaw), quality (brightness and sharpness), and confidence value (indicating the level of confidence that the bounding box contains a face). The response also provides a similarity score, which indicates how closely the faces match. By default now, only faces with a similarity score of greater than or equal to 80% are returned in the response. If needed, we can reduce this value by specifying the SimilarityThreshold parameter, as shown in below code, to give a higher tolerance when the target is blurry.

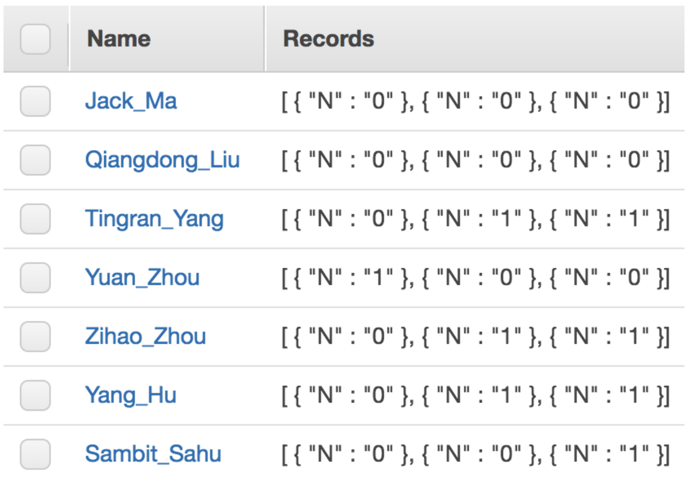
```python
def compare_faces(key1, key2, bT, bS):
    # retrieve and resize images for display
    bucketTarget = bT
    bucketSource = bS

    # Compare faces using Rekognition

    ret = rekognition.compare_faces(
        SourceImage={
            "S3Object": {
                "Bucket": bucketSource,
                "Name" : key2,
            }
        },
        TargetImage={
            "S3Object": {
                "Bucket": bucketTarget,
                "Name" : key1,
            }
        }
        # SimilarityThreshold=    the default is 80%, you can change it here.
    )
    return(ret)
```

With the result returned from each loop, in this stage, lambda will also update DynamoDB table by calling the UpdateItem API to input an entry which includes the name of people in the source photo and his or her corresponding compare result as a label. Intuitively, after completing the loop comparison, everyone will be labeled with number 0 or 1, where "0" represents didn't attend the class while "1" means attended. As shown in the code below, we use **list_append** method to edit the existing item's attributes, namely directly append label "0" or label "1" to the end of the field of people's **Records** column in DynamoDB table. In this way, we can always keep a history records of attendance of each student in this table, which makes us conveniently export the newly-come record and the corresponding summary code in stage 3. The updated DynamoDB table after stage 2 is shown in figure 5.

```
def update_index(ppName,cc,tablename):
    res = dynamodb.update_item(
        TableName = tablename,
        Key={
            'Name': {'S': ppName}
            },
        ExpressionAttributeValues={
            ":my_value":{"L": [{"N":cc}]}, ':r': {'L':[]}
        },
        UpdateExpression="SET Records = list_append(if_not_exists(Records, :r), :my_value)",
        ReturnValues="UPDATED_NEW"
        )
```



Name	Records
Jack_Ma	[{ "N" : "0" }, { "N" : "0" }, { "N" : "0" }]
Qiangdong_Liu	[{ "N" : "0" }, { "N" : "0" }, { "N" : "0" }]
Tingran_Yang	[{ "N" : "0" }, { "N" : "1" }, { "N" : "1" }]
Yuan_Zhou	[{ "N" : "1" }, { "N" : "0" }, { "N" : "0" }]
Zihao_Zhou	[{ "N" : "0" }, { "N" : "1" }, { "N" : "1" }]
Yang_Hu	[{ "N" : "0" }, { "N" : "1" }, { "N" : "1" }]
Sambit_Sahu	[{ "N" : "0" }, { "N" : "0" }, { "N" : "1" }]

Figure 6: A updated DynamoDB table after stage 2

3.2.3 Stage 3

When it comes to stage 3, Lambda function has already updated the corresponding table in DynamoDB. Then in this stage, Lambda will automatically call the **scan** API of DynamoDB to create a csv file from updated entries back in S3. The Scan operation returns one or more items and item attributes by accessing every item in updated table. Finally, a csv containing newly-come record and a history csv file will be put in statistic folder in S3, which will then be used by frontend to display.

```python
def createCsv(tableName, bucket, key1):
    try:
        t = time.strftime("%Y-%m-%d_%H-%M-%S",time.gmtime())
        table = dynamodb2.Table(tableName)
        response = table.scan()
        print ("List in table ", response['Items'])
        temp = key1.split('/')[:-2]
        key = '/'.join(temp)
        object = s4.Object(bucket, key + '/statistic/%s.csv'%t)
        object2 = s4.Object(bucket, key + '/statistic/history.csv')
        summary = "Name, Presence\r\n"
        file = "Name, Presence\r\n"
        # object.put(Body = header)
        for item in response['Items']:
            try:
                file = file + (item['Name'] + ',' + str(item['Records'][-1]) + '\r\n')
                summary = summary + item['Name'] + ',' + str(sum(item['Records'])) + '\r\n'
            except Exception as e:
                print("No records:" + e)
                continue
        print("Content in time.csv is ", file)
        print("Content in history.csv is ", summary)
        object.put(Body=file.encode('ascii'))
        object2.put(Body=summary.encode('ascii'))
    except Exception as e:
        print(e)
```

4 Result

In this section, a running test is implemented on our App to better show the process of uploading images and see the result of the image recognition and processing workflow.

Firstly, user needs to sign in or sign up before uploading photos in our App, the login interface as shown in figure 7. After the user sign up successfully, a private folder will be created to contain all files of the user.

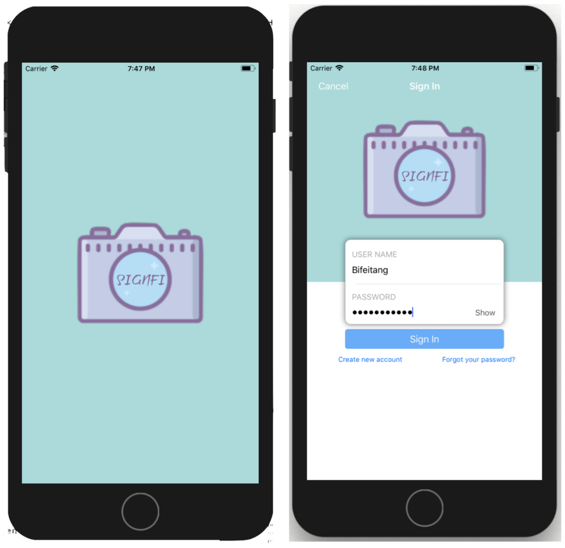


Figure 7: signin/signup interface

The second step is jumping to upload interface to upload user's source photos into his or her

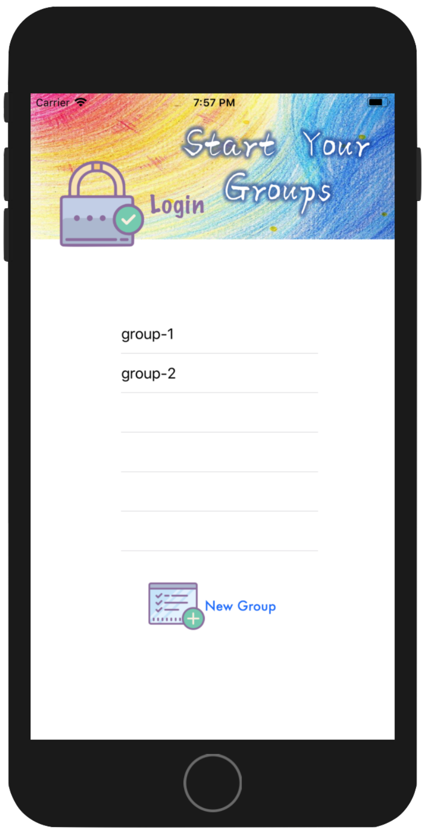


Figure 8: different folders of one user

private source folder. User can create different folders to contain various collections of people. For example, like in figure 8, a professor can create 2 folders named group 1 and group 2 for his or her different courses at the same time. At present, pictures in source folder are manually uploaded by us directly from back end, but we will integrate this function to the frontend soon.

When user wants to upload one or several target photos, it comes to the third step. Like shown in the figure 9, we firstly took a live photo and then uploaded it through our App. After waiting for a while, we will get the attendance list.

The final step is to get the return csv file and display it on our App interface. From figure 10 we could see, we firstly get different csv files named with various timestamp on the left screenshot. If we touch the newly-come one, the screen will jump to the right side screenshot. As we can see, flower is not in our uploaded photo just now, thus, the animation showed us with a "$\times$". In contrast, those people who showed in the uploaded photo are labeled with "$\surd$".

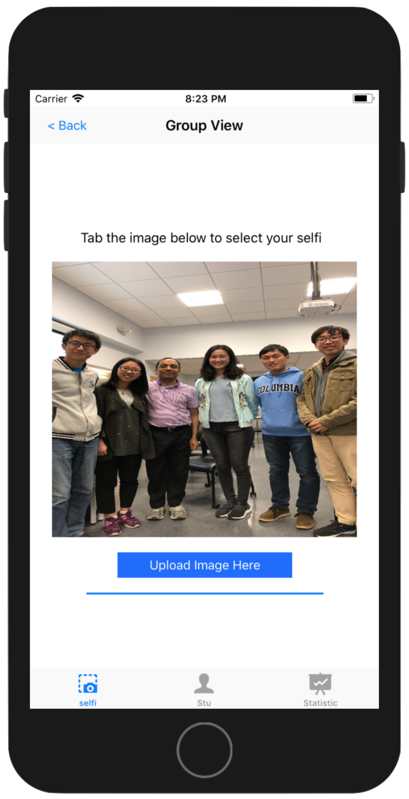
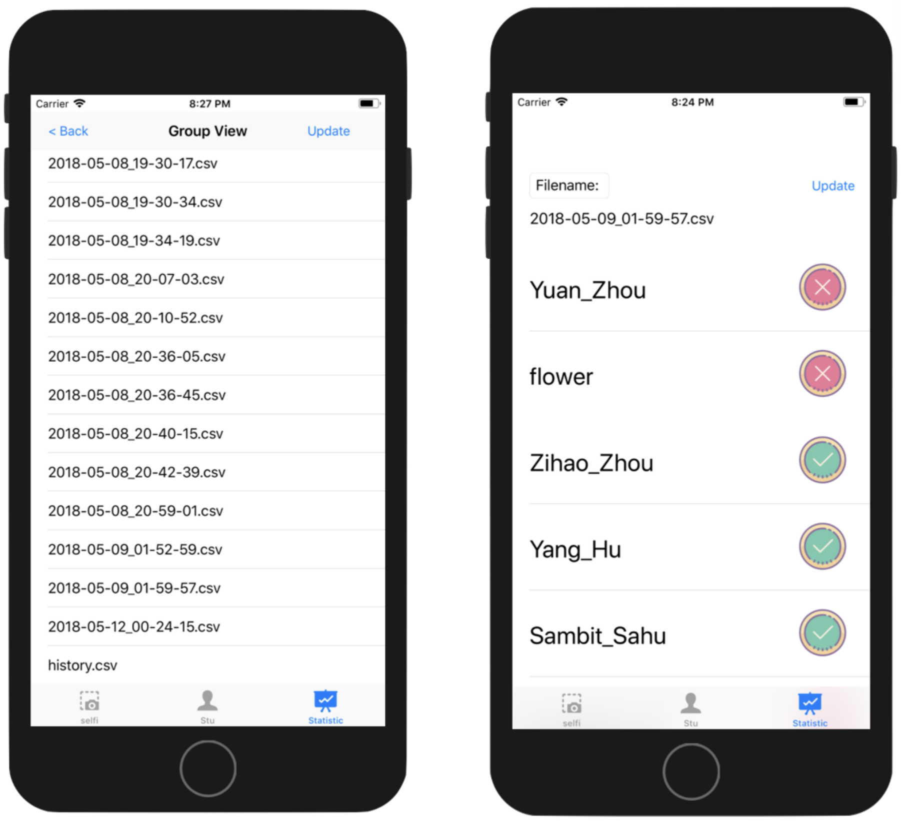


Figure 9: upload target photo



2018-05-08_19-30-17.csv
2018-05-08_19-30-34.csv
2018-05-08_19-34-19.csv
2018-05-08_20-07-03.csv
2018-05-08_20-10-52.csv
2018-05-08_20-36-05.csv
2018-05-08_20-36-45.csv
2018-05-08_20-40-15.csv
2018-05-08_20-42-39.csv
2018-05-08_20-59-01.csv
2018-05-09_01-52-59.csv
2018-05-09_01-59-57.csv
2018-05-12_00-24-15.csv
history.csv

Filename: 2018-05-09_01-59-57.csv

Yuan_Zhou	✗
flower	✗
Zihao_Zhou	✓
Yang_Hu	✓
Sambit_Sahu	✓

Figure 10: check the return result

5 Future work

Based on the work we have already done, we could improve the functionality in the future.

Firstly, we would integrate video face recognition model using Rekognition.[3] In this feature, people would be able to upload a short video to the cloud and get a return of presence or absence of the people in source. This could be a better solution for professor to deal with a large number of students for large class.

Moreover, scalability is significant for reproduction. we would test our APP on multiple ios devices to reduce potential compatibility problem, and to make sure AWS could handle large amount of requests at the same time.

In addition, we could release our APP on App Store where we could download on ios devices. We attempted to do so before Presentation, but ended to find out it takes weeks to review a new published APP until then we could publish it. We might also face a financial bottle neck since we run our APP on Amazon Cloud. It would be costful if a lot of users download our APP.

References

[1] *face recognition service using amazon.* https://aws.amazon.com/cn/blogs/machine-learning/build-your-own-face-recognition-service-using-amazon-rekognition/.

[2] *lambda refarch image recognition.* https://github.com/aws-samples/lambda-refarch-imagerecognition.

[3] *find distinct people in a video with amazon rekognition.* https://aws.amazon.com/cn/blogs/machine-learning/find-distinct-people-in-a-video-with-amazon-rekognition/.