Read Me

Jihan Li (jl4346), Mahd Tauseef (mt2932)
Xiaofan Yang (xy2251), Xiaochen Wei (xw2353)

Our app is an iOS-based application. There are three parts of code: front end, back end, and database.

Front end:

✧ Image Filter:

Basically we do Gaussian blur to the background image to improve the viewing effect.

```
CIFilter *gaussianBlurFilter = [CIFilter filterWithName: @"CIGaussianBlur"];
[gaussianBlurFilter setValue:clampFilter.outputImage forKey: @"inputImage"];
[gaussianBlurFilter setValue:@30 forKey:@"inputRadius"];
```

We use affine-clamp filter to stretch the image so that it does not look shrunken when gaussian blur is applied.

```
CGAffineTransform transform = CGAffineTransformIdentity;
CIFilter *clampFilter = [CIFilter filterWithName:@"CIAffineClamp"];
[clampFilter setValue:inputImage forKey:@"inputImage"];
[clampFilter setValue:[NSValue valueWithBytes:&transform objCType:@encode(CGAffineTransform)] forKey:@"inputTransform"];
```

✧ Regex Matching:

We use regular expression to validate the user's name and email address.

```
NSString *nameRegex = @"[a-zA-Z][a-zA-Z0-9]*";
    NSPredicate *pred1 = [NSPredicate predicateWithFormat:@"SELF MATCHES %@", nameRegex];
    NSString *emailRegex = @"^[_A-Za-z0-9-+]+(\\.[_A-Za-z0-9-+]+)*@[A-Za-z0-9-]+(\\.[A-Za-z0-9-]+)*(\\.[A-Za-z]{2,4})$";
    NSPredicate *pred2 = [NSPredicate predicateWithFormat:@"SELF MATCHES %@", emailRegex];
```

✧ Asynchronous Image Downloading:

To speed up the process of scrolling the table, we do not download all the head images at the beginning, but download them as the user views the page. We build an image cache in which the key is image url and value is image itself to be showed in the table view. Initially all the values are a default image. Then we use NSOperations to do concurrent downloading of the real images. Once an image is downloaded, we will change the value of the corresponding key to this image in the cache, and the table view will show it immediately.

```
[[NSOperationQueue mainQueue] addOperationWithBlock:^{
```

```
                UITableViewCell *updateCell = [tableView
cellForRowAtIndexPath:indexPath];
                if (updateCell)
                    updateCell.imageView.image = image;
            }];
```

✧ Google Map:

We use CLLocation to manage the location services of iOS, such as getting the location of user, tracking the movement of user. We show the route using Google Directions API, which gives the indeed route for walking.

```
mapView = [GMSMapView mapWithFrame:CGRectZero camera:camera];
DirectionService *mds=[[DirectionService alloc] init];
        SEL selector = @selector(addDirections:);
        [mds setDirectionsQuery:query
                    withSelector:selector
                    withDelegate:self];
GMSPath *path = [GMSPath pathFromEncodedPath:overview_route];
    GMSPolyline *polyline = [GMSPolyline polylineWithPath:path];
```

✧ Data Transfer:

We send request from the front end to the back end, and get the response from the back end using iOS request system.

```
+ (NSDictionary *)requestObjectWithURL:(NSString *)urladdress httpMethod:(NSString
*)httpMethod params:(NSDictionary *)sendParams
```

Back end:

✧ Server setup:

The backend code is present in /server directory.

In the root server folder we have the package.json file which includes all the package dependencies of the server. The app.js initializes the app and takes care of a few configurations.

The node_modules contains the npm modules that are required to run our code.

The majority of code logic resides in /server/router.

The file /server/router/index.js indexes all the basic URIs that the front-end uses to send its requests to the appropriate backend path.

Inside /server/router/routes we have the code for all the request handlers; there is one file for each of the basic URIs. In some cases, one file has code for multiple endpoints if the two endpoints serve a similar function. So for example, to view basic site info the URI is myAppIP/site and to view detailed site info the URI is myAppIP/site/detail.

The server receives the requests and routes them to the appropriate handler. Once there, the request body is searched for any required variables/information that that particular handler needs to perform its task.

The task is executed and a response is generated. A success is sent either by sending the data that the request needed or by sending the code 1. In case of error the code 0 is sent along with an error message if needed

✧ Routing:

We take the interesting site list obtained from the site section in front end as the input. Then we define an effective distance to reorder the sites in the list. In each round, we select out the site with minimum distance and put it into the list. Then take the location of the site as the new location of the user and recalculate the effective distance of other sites. We will also update the time by adding the traffic time and the typical tour time of this site to current time. Repeat this process until all the sites in the original list is either selected or taken off. We typically take a site off the list when the current time plus traffic time is not within the opening time of that site.

The equation for calculation effective distance is:

$$effectiveDistance = \|sitePosition - currentPosition\|_2 \times \frac{5.0}{popularity}$$
$$\times \frac{(currentTime < openTime)\,?\,(openTime - currentTime) : (closeTime - currentTime - trafficTime)}{endTime - currentTime}$$

✧ Ranking:

For user who has already show his interest of some sites, we will store his interest in the database as (User ID, Site ID, Score). For now, the score is set to 1 as long as the user once selects the site as his interest. After a certain period of time, we will transform the data into csv form, run a simple shell script to ssh into the Amazon AWS EMR Hadoop cluster, and run item-based collaborative filtering algorithm using Apache Mahout framework. The basic idea of this algorithm is to find the similarity of sites based on the interest history of all users. So we will rank the sites with high similarity to his interest history on the top of the list.

Database:

We use Java to crawl information of sites and events from Wikipedia, Tripadvisor and NYCgo website. Information of users, notes and interests are created by users. We use phpMyAdmin to load data into our database built on Amazon RDS (MySQL).

✧ Data structures:

We design the following five tables for our application:

Sites: basic info (name, photo), location info (city, address, zip code, latitude, longitude), contact info (phone number, website, email), description (history, culture, architect), travel notes (open time, recommending visit time, fee, activities) and ranking.

Events: basic info (name, photo), location info (city, address, zip code, latitude, longitude), contact info (phone number, website, email), travel notes (open/end date), brief introduction and full description.

Users: basic info (name, password, email, head photo) and profile (description, preference).

Notes: title, written time, clicked time and access control.

Interests: user ID, site ID, score.

We provide search functions for all five tables. Additionally, For Users, we provide insert and update functions. For Notes and Interests, we provide insert, delete and update functions.