



Munchen

Sana Sheikh & Connie Shi
iOS Programming
Fall '15

Introduction

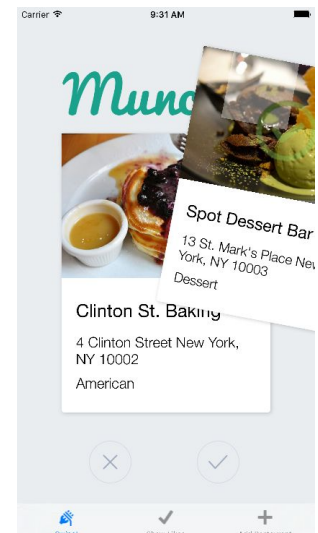
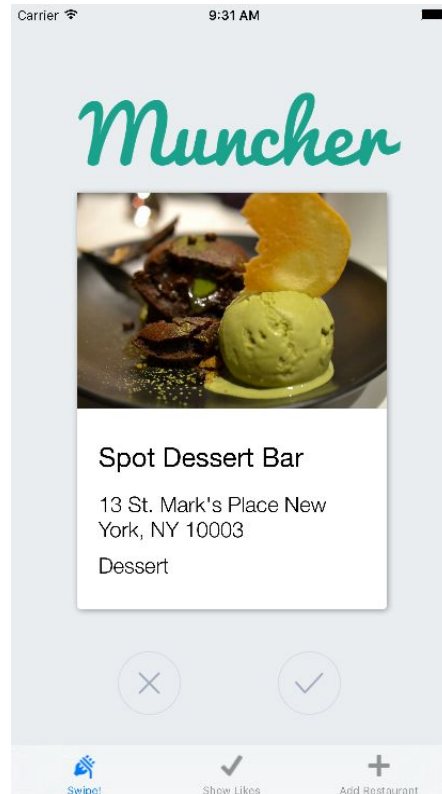
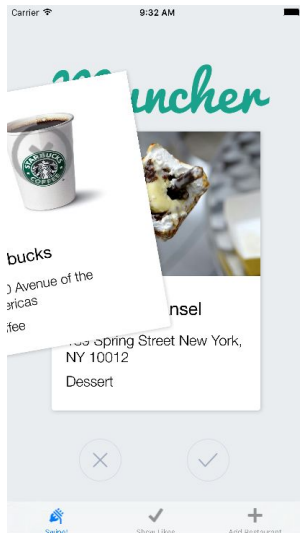
Living in a major city such as New York City can be both a blessing and a curse. With hundreds of restaurants and a plethora of cuisines, it can easily become burdensome to pick a place to eat. While web applications like Yelp certainly help narrow down choices by displaying customer ratings and reviews, users usually use this app with a certain restaurant in mind as opposed to browsing through new and interesting restaurants. This is why Muncher was created.

This app is designed to bring restaurant choices to users in a convenient way where they are able to like or unlike a given dining option with the swipe of a thumb. Muncher makes eating out not only easy but exciting – not only are established dining institutions provided, but also, any pop up or small business serving gourmet dishes from their personal kitchen get exposure through this app; we have given users the option to submit themselves as “restaurants” in order to promote their personal brands and provide a greater sense of community in local neighborhoods.

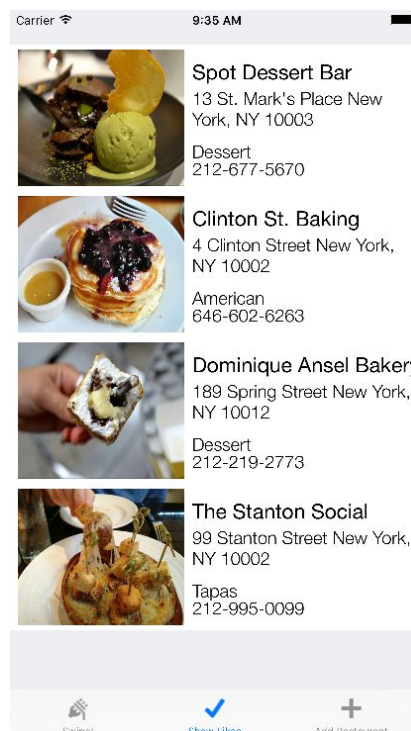
High Level Overview

Muncher has been designed with simplicity and convenience in mind to yield a more seamless user experience. The app consists of three tabs – swipe, show likes, and add restaurants.

The first tab displays several cards of restaurants stacked atop one another, with only the top card’s information and image being displayed at the front. The user then swipes left for an unlike and a right for a like. He or she can also press the “X” and “✓” buttons located just beneath the current card. As the user is swiping left or right, a shaded image of an “X” or “✓” will appear above the image on the card to ascertain the user’s final validation. Any unliked restaurants will be removed from the current view to reveal the next card underneath and will be marked as “seen” so the user will never encounter this restaurant choice again. This allows for more variety in options and overall better user experience. Any liked restaurants will be saved in the next tab, likes.



This tab displays a list of saved dining locations the user has previously liked, so to provide a centralized location for future reference. Each entry has an image and basic information and can be deleted with a left swipe. Once this option has been removed, again, it will be marked as “seen,” which will prevent it from being displayed in the card stack of restaurant choices in the previous swipe screen.



The last tab provides users the functionality to add restaurants. This view has several input fields, such as name, address, cuisine, price, number, and an image the user can upload. If any of these fields is left blank, an error notification will appear. Once users have fully input all information, a success notification come into view, the data will be saved into Parse, and all user-entered text will clear to allow for the submission of another restaurant, if desired. When users submit a dining option, because it is entered into Parse, anybody swiping restaurants in Muncher will be able to see that new restaurant as a viable option in the swipe tab.

Carrier 9:35 AM

Add Restaurants

Name

Address

Cuisine

Price

Phone

Select Photo

Sign Up

Swipe! Show Likes Add Restaurant

Carrier 9:35 AM

Add Restaurants


Name

Address

Cuisine

Price

Phone



Select Photo

Sign Up

Swipe! Show Likes Add Restaurant

Carrier 9:35 AM

Add Restaurants

Name

Address

Cuisine

Price

Phone

Missing Restaurant Info!

Please input all fields for the new restaurant before submitting.

OK

Select Photo

Sign Up

Swipe! Show Likes Add Restaurant

Carrier 9:35 AM

Add Restaurants

Name

Address

Cuisine

Price

Phone

Submitted!

Added new restaurant. Swipe or add another!

OK

Select Photo

Sign Up

Swipe! Show Likes Add Restaurant

Technical Aspects: Swiping Tab

One of the most important aspects we have utilized in Muncher is anonymous users. This is a built in Parse feature that bypasses sign up or login and automatically creates an account for the users the first they are using the app and any data changed or saved in this session will be maintained in future usages. The decision to implement this feature was to provide a more seamless, minimalistic user experience where most of the work is done behind the scenes. In the following method in the AppDelegate.m file, we have added a commented logout method used for testing purposes in order to clear any previously saved data associated with the current user and create a new account entirely. The method does the aforementioned or logs in to the already existing account for this user and returns a success boolean statement upon termination. Any error handlers are printed to the screen.

```
@interface AppDelegate ()
@end

@implementation AppDelegate

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    [Parse enableLocalDatastore];

    [Parse setApplicationId:@"tc60GyWzNcYXbGPdczXTWdq8yi942HWpTSTbtftI"
        clientId:@"RejBqE8K7uZBz6jPDVR1VySJV17xtPVN9fMV4681"];

    // Testing purposes to log out the user
    // [PFUser logOut];

    // Login capability with Parse's Anonymous Users
    if (![PFUser currentUser]) {
        [PFAnonymousUtils loginWithBlock:^(PFUser *user, NSError *error) {
            if (error) {
                NSLog(@"Anonymous login failed.");
            } else {
                NSLog(@"Anonymous user logged in.");
            }
        }]
    }
    return YES;
}
```

Another vital feature to the Swipe tab was swiping capability. This was constructed from the files DraggableView and DraggableViewBackground. The first file is the UIView for the "X" and "✓" buttons that are located beneath the restaurant card that, in addition to swiping, provide a way to like or skip the current restaurant. Next, DraggableView is the UIView of the draggable cards. Each frame displayed to the user is initialized with a restaurant's name, address, cuisine, and image and set up as subviews. The most important part of this view is the gestureRecognizer method, which is called when a user moves his or her finger across the screen. This method begins by extracting the coordinate data from the swipe movement and checking the current state the swipe is in, either the very beginning of the gesture, or in the middle of the swipe. In the latter case, a variety of minute changes in rotation, height, and scale are executed and applied to the overview, as shown below, which is a part of the

gestureRecognizer method. There are three states that the gestureRecognizer method can register: just started swiping, middle of swiping, and letting go of card. Each has its own actions associated to transform the draggable view.

```
// Checks what state the gesture is in
switch (gestureRecognizer.state) {

    // Just started swiping
    case UIGestureRecognizerStateBegan:{
        self.originalPoint = self.center;
        break;
    };

    // In the middle of a swipe
    case UIGestureRecognizerStateChanged:{

        // Dictates rotation (see ROTATION_MAX and ROTATION_STRENGTH for details)
        CGFloat rotationStrength = MIN(xFromCenter / ROTATION_STRENGTH, ROTATION_MAX);

        // Degree change in radians
        CGFloat rotationAngel = (CGFloat) (ROTATION_ANGLE * rotationStrength);

        // Amount the height changes when you move the card up to a certain point
        CGFloat scale = MAX(1 - fabsf(rotationStrength) / SCALE_STRENGTH, SCALE_MAX);

        // Move the object's center by center + gesture coordinate
        self.center = CGPointMake(self.originalPoint.x + xFromCenter, self.originalPoint.y + yFromCenter);

        // Rotate by certain amount
        CGAffineTransform transform = CGAffineTransformMakeRotation(rotationAngel);

        // Scale by certain amount
        CGAffineTransform scaleTransform = CGAffineTransformScale(transform, scale, scale);

        // Apply transformations
        self.transform = scaleTransform;
        [self updateOverlay:xFromCenter];

        break;
    };

    // Let go of the card
    case UIGestureRecognizerStateEnded: {
        [self afterSwipeAction];
        break;
    };

    case UIGestureRecognizerStatePossible:break;
    case UIGestureRecognizerStateCancelled:break;
    case UIGestureRecognizerStateFailed:break;
}
}
```

The remaining methods in this file deal with display of the card when it has actually been dragged to the left or right margin, so much so that it surpasses an action threshold and is designated as a complete right swipe or left swipe. Similarly, there are two equivalent methods but instead of being called during the dragging motion, it is called when the user presses the “X” or “✓” button located under the restaurant view.

The last file in this tab is DraggableViewBackground, which is the UIView that holds the DraggableView and is the main view in the view controller. When the current frame is initialized, all the restaurants must be displayed from Parse. We created a callback function to get all the restaurants and load the cards upon successful completion. This asynchronous function retrieves all unseen restaruatns from the

database by querying all restaurants that doesNotMatchKey of the seen relation. In this scheme, every time a user swipes left or right on a restaurant, it will be added to the user's "seen" relation. We only display restaurants that the user hasn't encountered before. The method itself, getRestaurants is displayed below:

```
// Callback asynchronous function that retrieves all unseen restaurants from the database
// By querying all restaurants and doesNotMatchKey in the seen relation
- (void) getRestaurants: (void (^)(void))completion {
    PFUser *user = [PFUser currentUser];

    // Get all restaurants and query it against seen relation
    PFQuery *queryAll = [PFQuery queryWithClassName:@"Restaurant"];
    PFRelation *relation = [user relationForKey:@"seen"];
    PFQuery *querySeen = [relation query];
    [queryAll whereKey:@"objectId" doesNotMatchKey:@"objectId" inQuery:querySeen];

    // Retrieve all restaurants that have not been seen before
    [queryAll findObjectsInBackgroundWithBlock: ^(NSArray *objects, NSError *error) {
        if (!error) {
            restaurants = [[NSMutableArray alloc] initWithArray: objects];
            completion();
        } else {
            NSLog(@"noooo :(");
        }
    }];
}
```

This callback is an asynchronous function that retrieves all unseen restaurants from Parse, which is done by querying all the restaurants and doesNotMatchKey in the seen relation. After obtaining all viable choices, a call to loadCards is made. As the name suggests, it creates a DraggableView object at a given index for every unseen restaurant. A minute but interesting design detail is highlighted in this function - for all the loaded cards, only display a fraction of the total cards at a time to provide a clean, aesthetically pleasing view of the stack of restaurant cards.

The other significant methods of this view is cardSwipedLeft and cardSwipedRight. Both are similar in their initial execution: the swiped card is removed from the loadedCards array and a subview is loaded of the restaurant underneath this card. However, the difference in the two functions is the relation that is stored in Parse. Left swiped cards are merely marked as "seen" so they will not be loaded from the getRestaurants callback function from earlier. Right swipe cards face the same fate in addition to saving the current restaurant and labeling it with a "likes" relation, which will be displayed in the next tab.

```
// Method adds the restaurant to either seen or likes relation
-(void)addToSeenOrLikes:(DraggableView*) card withRelation:(NSString*)column {
    PFUser *currentUser = [PFUser currentUser];
    PFRelation *seenRelation = [currentUser relationForKey: column];

    [seenRelation addObject: card.restaurant];
    [currentUser saveInBackgroundWithBlock:^(BOOL succeeded, NSError *error) {
        if (succeeded) {
            NSLog(@"Saved");
        } else {
            NSLog(@"Not saved");
        }
    }];
}
```

Technical Aspects: Show Likes

This view controller utilizes a UITableView to conveniently list previously liked dining options. When viewDidLoad is run, an asynchronous call to getLikes is made to retrieve only those restaurants this user has liked from Parse.

```
// Get only the restaurants that the user liked
-(void) getLikes: (void (^)(void))completion {
    likes = [[NSMutableArray alloc] init];
    currentUser = [PFUser currentUser];
    PFRelation *likesRelation = [currentUser relationForKey:@"likes"];
    [[likesRelation query] findObjectsInBackgroundWithBlock: ^(NSArray *objects, NSError *error) {
        if (!error) {
            likes = [[NSMutableArray alloc] initWithArray: objects];
            completion();
        } else {
            NSLog(@"noooo :(");
        }
    }];
}
```

From here, each cell for the UI Table View is populated with the restaurant name, address, phone number, and image and returned to be displayed. The outlets for each field is associated with a tag number and each field is populated by retrieving the tag number. Additionally, to provide greater functionality, users are able to swipe left on a cell to delete it from the list. This will still be marked as "seen" in Parse, so the user will not encounter this option in future swipes.


```

// Populate each cell with the restaurant information including...

// Load image to a default first in order for the image to show
UIImageView *imageView = (UIImageView*)[cell viewWithTag:100];
imageView.image = [UIImage imageNamed:@"loading.png"];

// Load name of restaurant
UILabel *name = (UILabel*)[cell viewWithTag:101];
name.text = [obj objectForKey:@"name"];

// Load the address of the restaurant
UILabel *address = (UILabel*)[cell viewWithTag:102];
address.text = [obj objectForKey:@"address"];

// Load the cuisine
UILabel *cuisine = (UILabel*)[cell viewWithTag:103];
cuisine.text = [obj objectForKey:@"cuisine"];

// Load the phone number
UILabel *phone = (UILabel*)[cell viewWithTag:104];
phone.text = [obj objectForKey:@"phone"];

// Load the photo of the actual restaurant obtained from PFObj
PFFile *photoFile = [obj objectForKey:@"image"];
[photoFile getDataInBackgroundWithBlock:^(NSData *data, NSError *error) {
    if (!error) {
        UIImage *photo = [UIImage initWithData:data];
        imageView.image = photo;
    }
}];

```

Technical Aspects: Add Restaurant

The last view controller gives users the opportunity to add restaurants to the database. It consists of six text input fields and an image view that accesses the user's photos to upload an image. If the user tries to submit the page with missing fields, an alert will notify him or her of the error. Once all fields are present and the user presses the submit button, a new restaurant PFObj is created in Parse with the specified attributes the user typed in. The data is saved asynchronously and an alert will state the successful submission. All text fields and image view will clear to allow the user to input another restaurant if necessary.

Users can choose a photo in their camera gallery to upload to Muncher. This uses UI Image Picker Controller to choose an image. They are given the option to view the photo then "Choose" or "Cancel".

```

// Allows the user to select a photo from the Photo Gallery
- (IBAction)selectPhoto:(id)sender {
    UIImagePickerController *picker = [[UIImagePickerController alloc] init];
    picker.delegate = self;
    picker.allowsEditing = YES;
    picker.sourceType = UIImagePickerControllerSourceTypePhotoLibrary;
    [self presentViewController:picker animated:YES completion:NULL];
}

// Picker Controller allows you to pick an image from Photo Gallery
- (void)imagePickerController:(UIImagePickerController *)picker
    didFinishPickingMediaWithInfo: (NSDictionary *)info {

    UIImage *chosenImage = info[UIImagePickerControllerEditedImage];
    self.photo.image = chosenImage;
    [picker dismissViewControllerAnimated:YES completion:NULL];
}

// Cancel option from picking a photo in photo gallery
- (void)imagePickerControllerDidCancel:(UIImagePickerController *)picker {
    [picker dismissViewControllerAnimated:YES completion:NULL];
}

```

We further perform error checking for fields that should be filled out but are instead left blank. If any of these fields have not been filled out or supplied, then an alert will be generated to which the user can click “OK” and given the opportunity to fill out the form properly. This error checking is necessary to ensure that there is data integrity.

```

// Check if all fields have been filled out to prevent errors
if (!restaurantName.hasText || !address.hasText || !cuisine.hasText
    || !price.hasText || !phone.hasText || photo.image == nil) {

    UIAlertController * alert = [UIAlertController
        alertControllerWithTitle:@"Missing Restaurant Info!"
        message:@"Please input all fields for the new restaurant before submitting."
        preferredStyle:UIAlertControllerStyleAlert];

    UIAlertAction* yesButton = [UIAlertAction
        actionWithTitle:@"OK"
        style:UIAlertActionStyleDefault
        handler:^(UIAlertAction * action) {
            [alert dismissViewControllerAnimated:YES completion:nil];
        }];

    [alert addAction:yesButton];
    [self presentViewController:alert animated:YES completion:nil];
    return;
}

```

Upon successfully filling out the form and submitting, we asynchronously `saveInBackgroundWithBlock` the uploaded image to the associated user and proceed to save the restaurant information as well. This persists the data to Parse for future use. Once this action is complete, every user on Muncher will be able to view this newly saved restaurant as part of their swipe cards.

```

// Save the image to Parse
[imageFile saveInBackgroundWithBlock:^(BOOL succeeded, NSError *error) {
    if (!error) {

        // The image has now been uploaded to Parse. Associate it with a new object
        [restaurant setObject:imageFile forKey:@"image"];

        // Save the restaurant to database asynchronously
        [restaurant saveInBackgroundWithBlock:^(BOOL succeeded, NSError *error) {
            if (!error) {
                NSLog(@"Saved");
                UIAlertController * alert = [UIAlertController
                    alertControllerWithTitle:@"Submitted!"
                    message:@"Added new restaurant. Swipe or add another!"
                    preferredStyle:UIAlertControllerStyleAlert];

                UIAlertAction* yesButton = [UIAlertAction
                    actionWithTitle:@"OK"
                    style:UIAlertActionStyleDefault
                    handler:^(UIAlertAction * action) {
                        [alert dismissViewControllerAnimated:YES completion:nil];
                    }];

                [alert addAction:yesButton];
                [self presentViewController:alert animated:YES completion:nil];
            } else {
                NSLog(@"Error: %@", error, [error userInfo]);
            }
        }
    }
}

```

Future Endeavors

While we aimed to have all our intended features completed, we were not able to under our given time constraints. Our next steps will be:

- Integrating geolocation (Google Maps API) and expansion to provide for several major cities
- Monetization → Have pop-ups and small business that establish themselves as restaurants pay a fee for using Muncher as a platform for promotion
- Allow messaging capability with restaurants to obtain more information
- Integrate Facebook login and sharing for greater exposure of Muncher
- Use factual API's restaurant scheme to provide all the established dining institutions in the United States, as opposed to manually inserting into Parse