Final Report

Location Based Photo Sharing Application

Student name: **Christopher Paterson**

Student number: **C1229143**

Supervisor: **Dr. Matthew J W Morgan**

Moderator: **Dr. Kirill Sidorov**

Module: **CM3203 – One Semester Individual Project**

Table of Contents

Introduction

The aim of this project was to create an iOS application to openly share photos that have been taken at a location. These photos can be tagged and given a description or explanation. Users around the area can look at these photos in reverse chronological order and then like and comment on them. By the end of the project the application is at a point where it could be launched as a minimum viable product.

Intended Audience

There are two ways of using the application, creating and consuming content. The intended audience for the creation of content side are those who enjoy sharing. This could be sharing their experience at a location or event or maybe they just want to share a photo they took. The second intended audience are those who will consume the content. These could be people who are interested in a particular place or city such as tourists. There will, of course, be overlap between these two areas but these are the intended audiences that have been identified.

Project Scope

The scope of the project has remained unchanged from the initial report. They are the features that I deem to be required if the application were to launch as a minimum viable product.

- Mobile
    - Ability to share, comment and tag users own photos.
    - Ability to comment and rate other users photos.
    - Ability to view photos taken near user's current location.
    - Intuitive interface and usage for users.
    - Ability to save photo to upload later for times with no network connectivity.
    - Ability to manage shared photos while not at location.
- Web
    - Administrator ability to remove inappropriate photos and comments.
- General
    - Include photos from Flickr to expand the photos base.

Background

Technology

The chosen platform for the application is iOS. This allows me to take advantage of the control Apple has over its devices hardware and software to provide a more consistent experience to the user as opposed to Android's countless devices and configurations. Another option would have been to use a tool such as Xamarin (Xamarin, 2016) which allows the developer to share code across iOS, Android and Windows phones. However, it is not as powerful as creating a native application and some code such as the user interface is not reusable which would have added a non-insignificant amount of development time to the project if I were to make it multi-platform.

The choice of languages for this project were between Swift and Objective-C, of which I decided to use Swift. Swift is a relatively new language made by Apple for use in iOS/OSX application development as well as general purpose use. Although it is a new language, it is growing very quickly in popularity rising from 24th to 15th place in the TIOBE index (TIOBE Index, 2016) in one year. It is also interesting to note that Objective-C has fallen from 4th to 13th place in the same time period so while it is still more popular than Swift, it appears that more people are moving over to Swift. There were several reasons for selecting Swift over Objective-C. The first one is that it is easier to read and learn and since I would be learning during development it feels more appropriate to use Swift. The second reason is that there is no real need to deal with memory management like there is in Objective-C. This allows for quicker development as well as reducing the likelihood of bugs/leaks.

Another large advantage that Swift has over Objective-C is the inclusion of optional types. An optional type is a way of explicitly stating that there may be no items or values returned (nil). These optional values can then be ‘unwrapped’ at a point in the code when the developer knows that there will be a value. This is especially important in this application as there is a lot of asynchronous activity such as retrieving photos from multiple sources. If something were to try to perform an action on a nil value then the application would crash. Due to the asynchronous nature of the application there are many places that a nil value may exists so having the use of optional values helps create an overall more stable application.

The downside of using Swift is how young the language is. Most examples will be written in Objective-C so it may be more difficult to get support on issues I may have. However, iOS makes use of the Cocoa Touch API, a framework for building applications to run on iOS to provide abstraction from the lower levels. Due to this, since the majority of examples will be interacting with Cocoa Touch API, a lot of the class names will be the same allowing me to use Objective-C examples to better solve the problem I am working on. Another downside of the language being so young is that it is still changing. Luckily, Xcode aids in updating code to match the newest version so this should not cause a large problem.

Overall, Swift is a simpler and cleaner language than Objective-C. Not only does the lack of header files help keep the project clean but, in general, less code is required to get the same results as Objective-C (Cristian González García et al., 2015).

The IDE chosen for development was Xcode. I had considered using JetBrains AppCode due to past experience with their products but with Xcode I would have access to the latest versions of both Swift and iOS (beta) sooner than AppCode. Similarly, with Xcode beta I am able to develop for the beta version of iOS.

Parse is a backend as a service that allows for a quick setup time as it is hosted on it's own server meaning the developer does not have to create both the database and the API to interact with the database. It is also quick and easy to modify as tables can be created via code, for example, if the developer needs to create a table to store likes, they are able to simply write the query that would persist the object and if the table does not exist it is created or if it does exist but a new field has been added to the query, the new field is added to the table. This is very useful during the early stages of development when not much is concrete and tables often need to be modified or created. As well as quick setup time, Parse also allows for a lot of useful functionality such as user creation/management, asynchronous fetching/saving and password hashing. Parse has software developer kits (SDK's) for many platforms including iOS/Swift and PHP which should provide for a relatively short setup time.

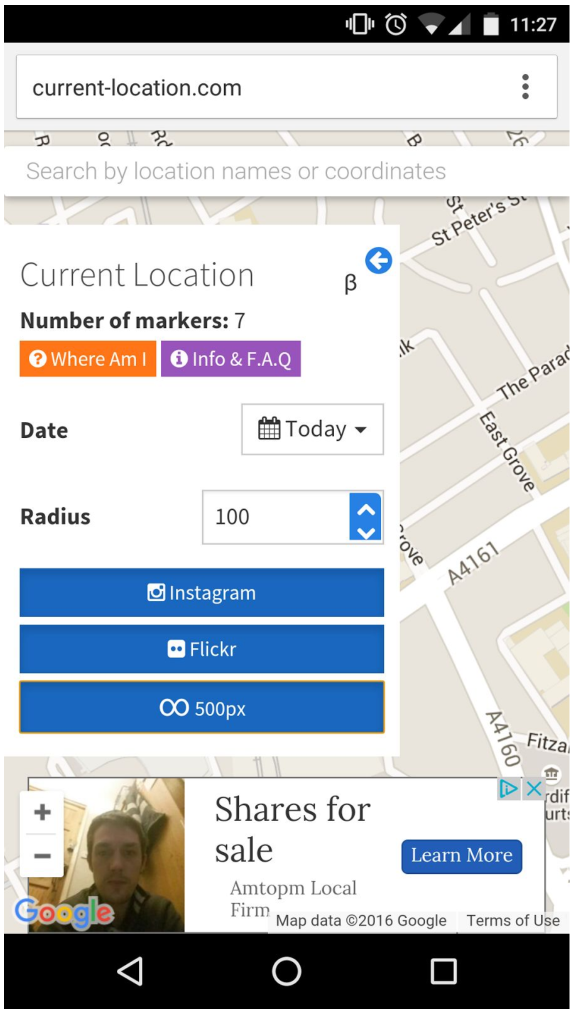
The web side of the project was made using Laravel 4.2.X, a PHP framework. Similar to Parse, Laravel provides a lot of useful functionality out of the box such as routing, sessions caching and integrated unit tests. These features allow the developer to focus on the web application they are building rather than the tools to create it. Laravel uses a Model-View-Controller (MVC) architecture to help with structuring a project. The The default Object Relational Mapper (ORM) is Eloquent however, this has switched this out for Parse as that is the databased used to store data collected from the iOS application. For the templating engine Laravel uses Blade which not only makes the HTML code cleaner and easier to read but provides protection against attacks such as XSS with minimal effort on the developer's side.

Flickr (Flickr, 2016) is an image and video hosting web service that relies on user generated content. This user generated content contains Exchangeable image file format (Exif) data which stores information about the device the photo was taken on. Exif data may include information about camera settings, date and time created and, most importantly for this application, geolocation. Flickr provides an API that allows developers to query for public photos using many different criteria including latitude, longitude, radius and date.

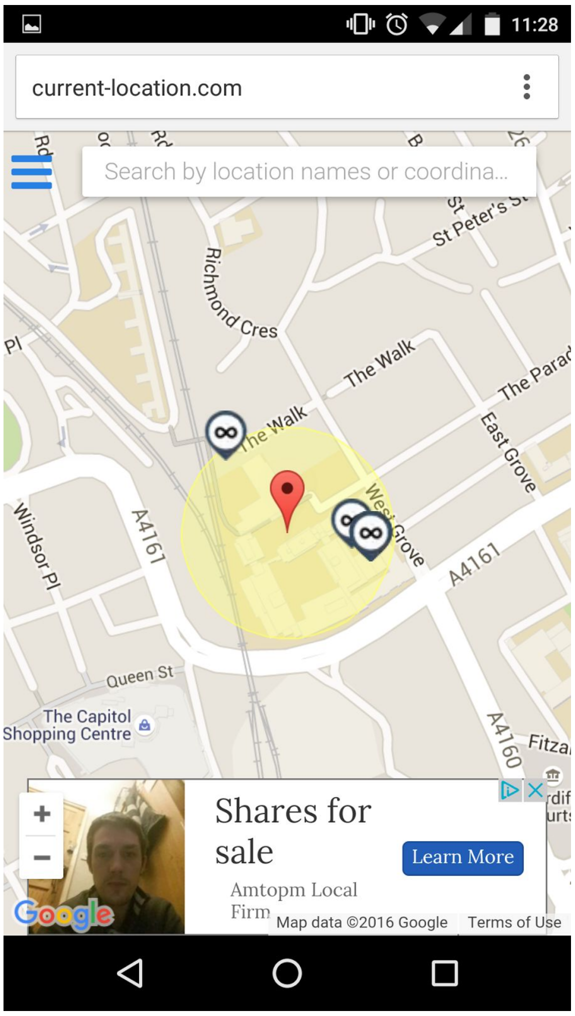
Existing solutions

Current Location

Current Location (Current Location, 2016) is a web application that aggregates photos from Flickr, Instagram and 500px within a user defined radius at the user's current location. The web application works well with mobile browsers allowing the user to use it while away from a computer.



current-location: options



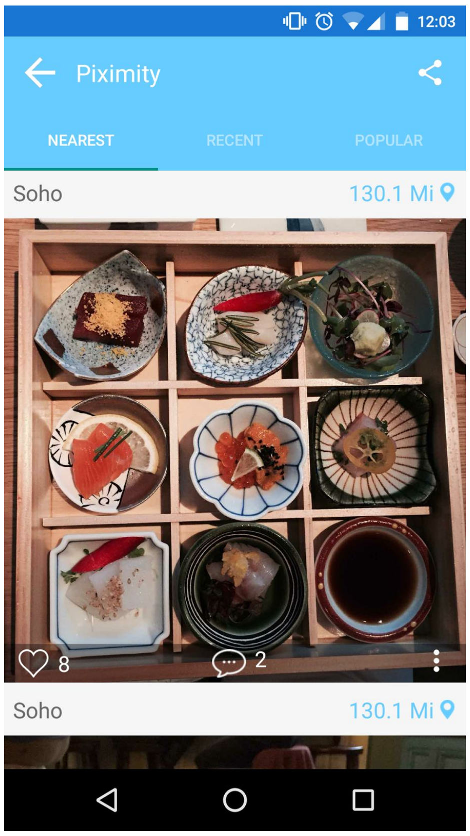
current-location: photos around location

Current Location provides the user with more control over what they see than the project application. Firstly it has more sources of data which the user can enable or disable as they wish. Secondly the user is able to control the search radius which is beneficial for different environments as the user may wish to have a smaller radius when in a building than in a public space such as a park.

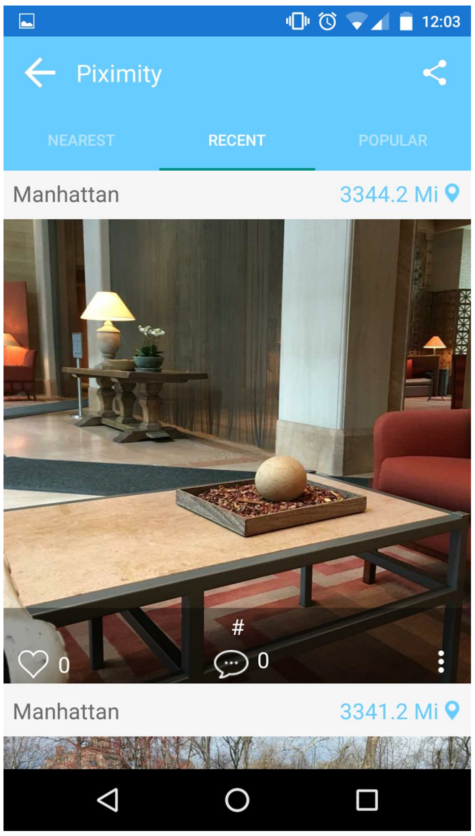
There are several differences between this application and the project one. The largest one is the lack of a social aspect. Current Location does not provide the opportunity to favorite or comment in place, however, the user is able to open a link to the photo on the hosting website and interact there. The second difference is the lack of ordering. Current Location displays the photos as pins in a Google map which the user can then click on to view them but it does not offer the ability to cycle through them chronologically. Another difference is that the user is able to drag the point to different areas of the map allowing view photos in any location meaning it is not tied to the users location.

Piximity

Piximity (Piximity, 2016) is an Android and iOS application that allows users to view photos shared by other users. Rather than providing a radius the user is able to sort by nearest and it displays the distance in miles. Users are also able to order by recent and popular. Similar to the project application, users are able to share, tag, like, comment, and report photos.



Piximity: Nearest photos



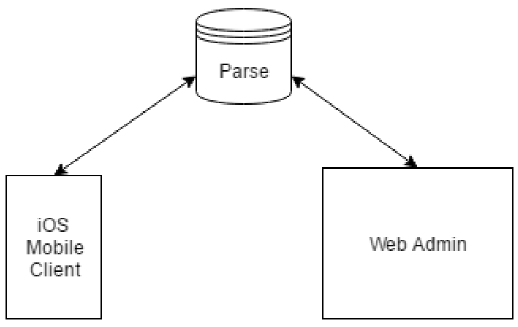
Piximity: Recent photos

The dataset for Piximity is very small. The nearest image to Cardiff is ~130 miles and was taken 1 year ago. With the project applications integration of Flickr, the hope is that users will always have content to view that is relevant and near them and as such will feel more inclined to use the application.

A useful feature that Piximity has is that it allows the users to view photos by events, e.g. Wimbledon 2015. This provides extra information about photos which could be valuable to users. As of yet, no such feature exists within the project application, however, it may be possible to create a grouping feature based on recurring tags within an area and timeframe.

Approach

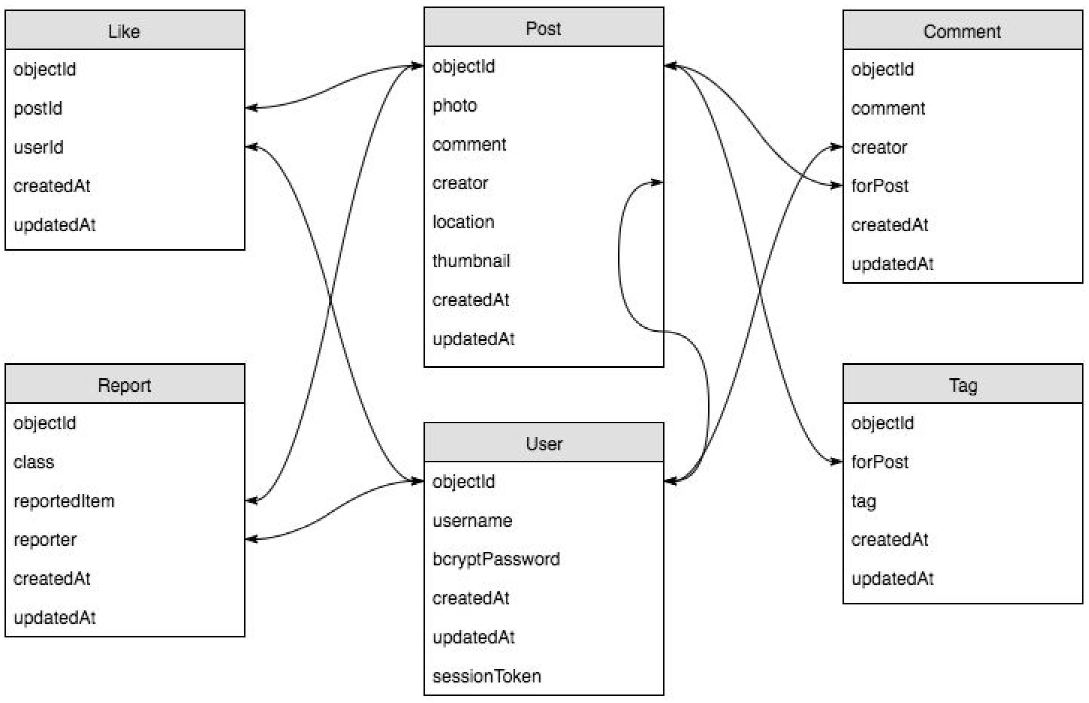
The overall architecture is simple. The mobile client queries and writes to the Parse database, as does the web administrator client. The mobile and web clients never directly interact with each other.



Architecture

Database

The database has a relational structure as the data lends itself to this rather than a non-relational structure. It would be possible to convert to a non-relational structure with Post being the highest level object, however, I have the most experience with relational databases and at the moment the application does not require the benefits such as scalability in such an early stage.



Parse database tables

Parse uses the device's local time to include a createdAt and updatedAt field for each table by default. This has been useful for ordering items and since the application works in a small area around the user and not globally this is acceptable.

The database was expanded and modified as needed along with the application with the tables being added in the order listed below.

User - This is the account that each user makes when signing up. It is used in almost all other tables to link back to an owner or creator of an object. Parse automatically hashes the passwords using bcrypt which adds an extra layer of security. Information such as email address, forename, and surname have been omitted as although they may be useful to have in the future, as of now it is an extra barrier for the user to sign up.

Post - This is the main object of the application. It stores the uploaded photo as well as a thumbnail, an optional description (comment) and the location the photo was taken. The location type is a Parse geopoint which stores latitude and longitude. It is possible to then query in a radius around this to get all posts around a given user location.

Initially the latitude and longitude were stored in addition to the geopoint. The reason for doing this was to allow the database to be switched out easier if needed. However, these were removed as it would be trivial to retrieve the latitude and longitude from the geopoint during a migration and keeping them in may cause confusion if new developers were to start working on the project.

Like - A user is able to ‘like’ a photo. The purpose of would be to provide another method of sorting photos if there was extra time at the end of development. It could also be used in conjunction with tags to create recommendations for places that users may wish to visit.

Comment - Users may comment on a photo. This allows other users to add value to a post by maybe adding more information about that area/event or simply complementing the photo.

At the moment comments are a flat structure on posts, i.e. you can not comment on comments. It would be preferable to redesign this table to be polymorphic, similar to the report table, to allow for this.

Tag - The main reason for including a tagging system was to create recommendations but the inclusion of tags also provides another way of describing photos outside of the creator’s comments which adds more value to them.

Tags are user decided as creating a comprehensive list of all tags users may wish to use, although simplifying recommendation and moderation, would be a very large task and is out of the scope of this project.

Report - Due to the nature of the internet and anonymity, there are sometimes people who will take advantage of this for malicious purposes. For this reason a report function has been added. It stores if it is a comment or post, the person who made the report as well as the item reported which can then be used to find the creator.

The initial plan was to have a column called 'item' and have a pointer to whatever item had been reported, however, Parse does not support pointers of different types in the same column which would mean there would have to be a column for each reportable item which is not so bad at this stage but would not be appropriate for scaling.

As of now, if a photo is deleted then the comments and likes are not deleted. The reason for this is because in the future, if soft deletion were to be implemented, it would make the database simpler and require less read/writes to reinstate a photo as there would be no need to update all the comments and likes to remove isDeleted. On top of this, the storage requirement for the comments and likes are very small in comparison to the photos and as such keeping them should not cause a large amount of wasted space.

The above database design was chosen because it attempts to keep the data modular which will help for future expansion. Tables are linked using objectIds which helps reduce the number of writes. For example, usernames are unique and as such could be used to link objects to the User table. However, If the application were to support changing usernames then every entry related to that user would have to be updated. This also helps prevent data anomalies if we were to forget to update something. In general each table contains the minimum amount of information to function.

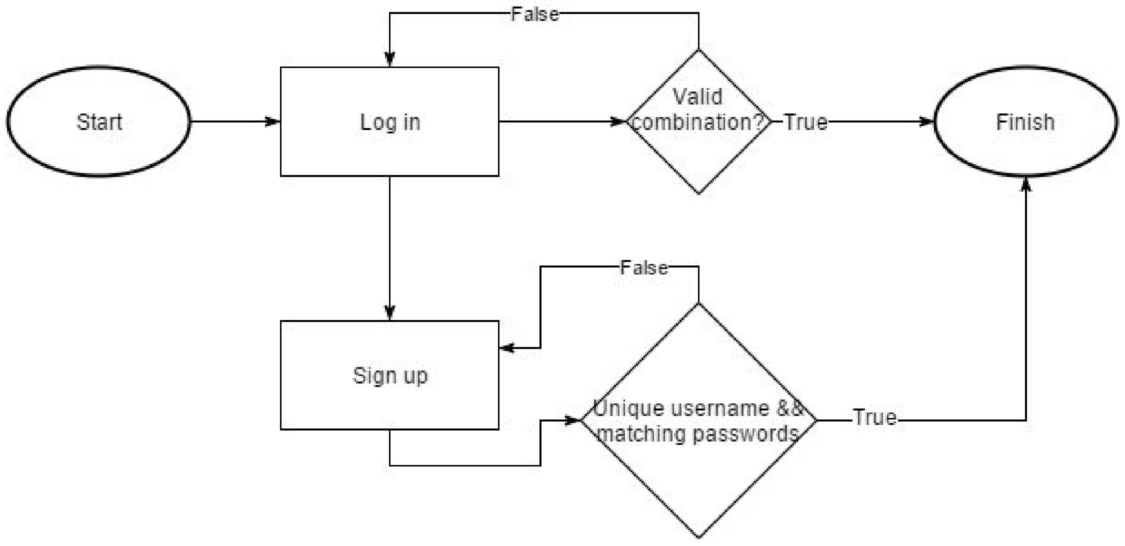
Another source of data is the Flickr API. Photos are pulled in from Flickr and displayed to the user using the same radius as native photos. Users are able to interact with these in much the same way as they can with photos shared from within the application with the only functionality missing being the ability to report a Flickr photo.

Mobile

The mobile application has been the main bulk of the project which is the only thing that users see. The aim has been to make this intuitive to the user so that they are able to complete the task they set out to with minimum friction.

Log in or Sign up - Upon starting the application for the first time or after logging out the user is met with a log in screen. If they have signed up before they can enter their credentials to log in or they are able to create a new account by tapping 'Sign up'. If there are any issues with logging in or the sign up process then the user is notified using an alert box. The reason for the log in screen being the initial one rather than the sign up screen is because over the lifetime usage of the application, it is most likely to be used the most within this flow.

After the user has logged in, subsequent application launches will take the user directly to the home page.



Login user flow

All user flows below this assume the user is logged in.

View photos around current location - This is the home view and the one that the user sees upon starting the application. The reason for this to be the first screen is so that the user is greeted with content. The content is split into two sections, those photos shared from within the application and those that have been found using the Flickr API. The reason for splitting these two into sections was largely for implementation reasons outlined in the implementation section but another reason would be because the use may wish to know the source of the image.

At the moment the images refresh when the user pulls the view down. Refreshing ever certain period of time or distance traveled may lead to a smoother user experience but the problem with this would be bandwidth use. Due to the nature of the application, users are likely to be using it on their mobile data and loading a possibly large number of images that the user may not want to see at that moment could lead to a large amount of data usage. However, since the application would only reload images while open, it could be argued that the user may want to see live updates and would not worry about the data usage.

Interact with a post - Once the user taps an image on the home screen they are taken to the ‘View Photo’ screen where they are able to interact with it. The image on the ‘View Photo’ screen is relative small, taking up less than 50% of the screen. Users are able to view the photo in full screen by tapping it which takes them to a view that displays it at a large size. On the full screen page they are able to zoom in up to 500%. Once they close this screen they are taken back to the ‘View Photo’ screen. The reason for choosing tapping the photo to make it fullscreen was because adding a ‘view fullscreen’ button would cause more clutter on the screen and it is highly likely that users will be able to find this feature just by using the application normally.

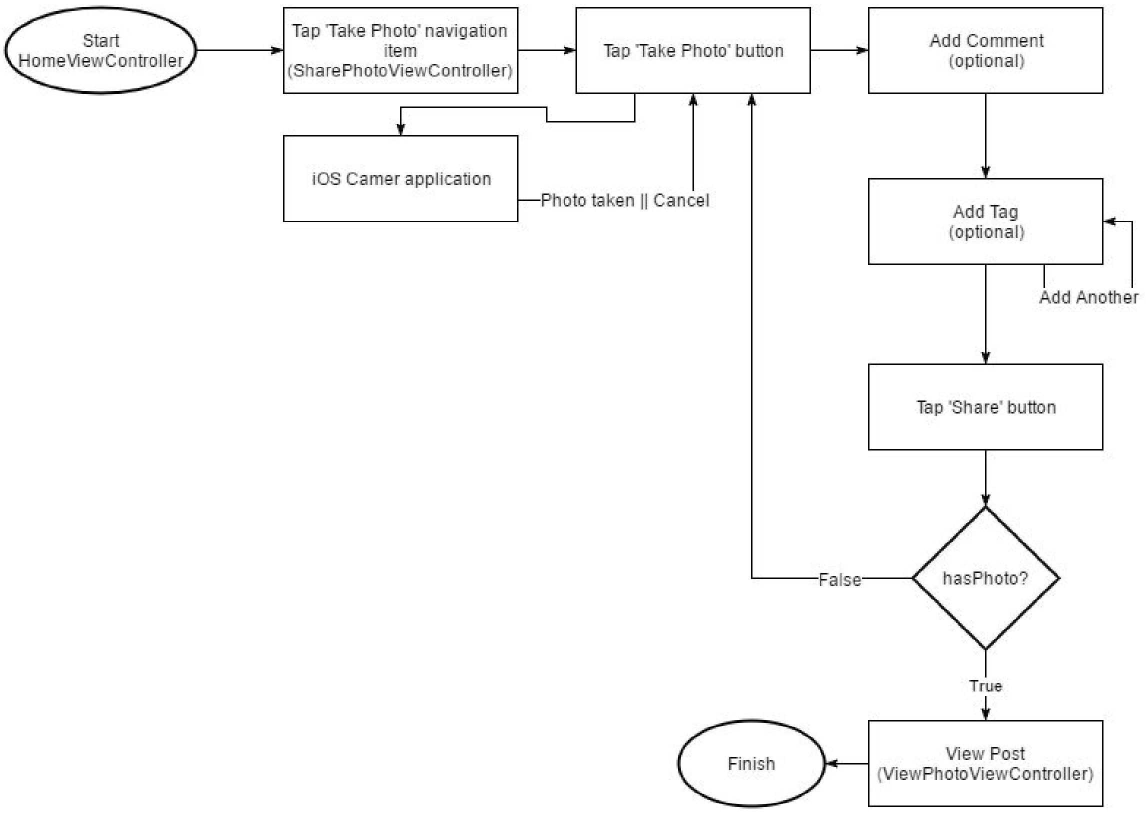
On the ‘View Photo’ screen the user is able to like the photo simply by tapping the heart icon. At the moment this serves no purpose but in the future it will be used to rate by popularity as well as to encourage users to share photos for a social reward. The reason for choosing a like system instead of a rating system is because it leaves less room for subjectivity. It is not possible to create an objective scale when it comes to this type of thing and as such one users 4/5 rating is not the same as another user’s.

Users are also able to comment on photos. At the moment there is not a character limit on comments, however, beyond a certain length they are cut off when displayed. Once the user taps the ‘Submit’ button on a comment, if the text field is not empty, the comment is added to the top of the comments table view.

Users are able to both comment and like photos from Flickr in much the same way they can with content shared from within the application.

Sharing a photo - The order in which the user takes the photo, comments or tags does not matter. At any point the user may tap the cancel button which resets the view. The only required item to share is the photo and if one has not been taken then the user will be unable to submit a post and an alert will tell them that ‘You must take a photo to share’. Tags are added one at a time. Once the ‘Add Tag’ button has been tapped, the tag is added below and can be removed by tapping it.

Providing there is a photo, once the user submits the post they are taken to the ‘view photo’ view to see it.



Submit photo user flow

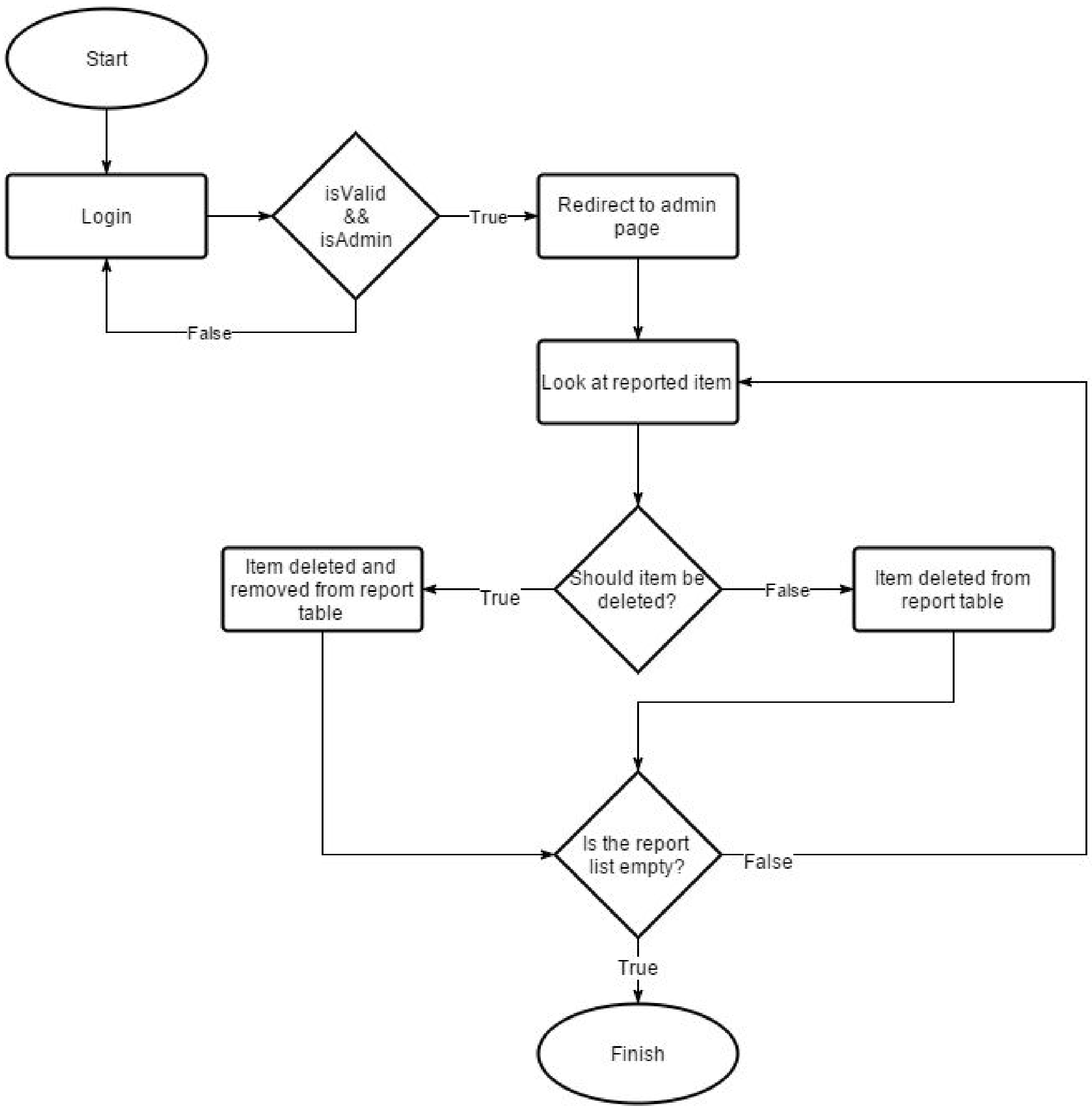
Report a post or comment - Users are able to report a photo or comment by long pressing the item they wish to report. After this they are met with an alert asking if they are ‘sure you wish to report this item as inappropriate?’. If the user presses ‘OK’ then the report is sent and the user is told that the report has been successful.

The issue with report at the moment is discoverability. Some users may not think to long press on certain objects and it’s not entirely obvious which objects are reportable.

Manage own posts - A user is able to delete photos they have shared without being at that location. To do this they go to the settings screen then tap ‘Manage Posts’. This takes them to a screen that displays all of their photos in a table view. They are then able to swipe left on the post they wish to delete and after conformation, it is deleted. The hope was to allow users to edit the comment and tags, however there was not enough time and as such the user is only able to delete a post. This is a feature that should be added later.

Web

The purpose of the administrator website is remove explicit or malicious content from the platform. If the goal of having the project at a point where it is ready to launch by the end then there must be a way of removing such content. The website is a very simple 2 page website. It has a login page and a tabbed page to view reported comments and posts.



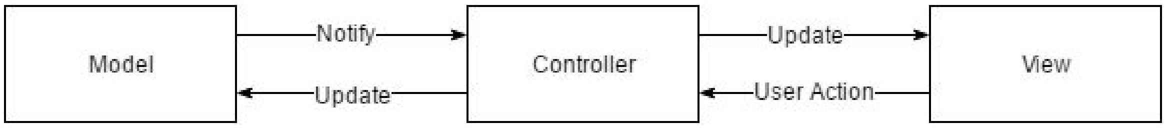
Administration of reported photos flow

The administrator website has been kept simple as the end user will not see it and time is better spent on what the user is able to experience rather than tools at such an early stage. As of now the website is not effectively able to support multiple administrators working at the same time due to data being refreshed on page refresh and not in real time. If there were multiple administrators working at the same time it would likely lead to duplication of work.

The comment and post tables are largely the same with the exception that the post table allows the administrator to click the 64x64 thumbnail to see a larger version in a modal.

If an administrator deems it necessary to remove a post or comment then they are met with a confirmation alert. This may become tedious over time but the extra step is preferable to accidently deleting a post that should not be deleted as deletion is currently irreversible. At the moment there is no audit trail of which administrators has done what action to what post/comment so administrators must be trusted.

Despite being a simple web application, the Laravel framework has been used to create it. Laravel uses a Model-View-Controller architecture which provides a lot of benefits including helping the developer design code that is modular and reusable which makes it easier to maintain and expand at a later date. In general, MVC helps the developer to manage complexity (Aku Kolu, 2012, pp. 15-16).



Model-View-Controller

An alternative to Laravel would have been Express, a framework for developing web applications using Node.js. I have some experience with the JavaScript language and integration with Parse would have been smoother as it is easier to swap out parts in Express than in Laravel where the default is the Eloquent ORM using SQL. However, I have more experience with creating web applications using PHP and Laravel and as such I judged that the time to get Parse setup and working with Laravel would be shorter than learning to use Express and Node.js.

The web application uses a RESTful API. The benefits of doing so is providing loose coupling which will help with scalability at a later date. Similarly, now that the web application has been created, if the project were to move away from Parse in the future, it would have a solid starting point for creating the API for the mobile application to interact with.

Testing

Initially the plan was to do test driven development, however, it would have been the first time I had done so for a project and as such would have taken too long to learn and implement. Both Swift and iOS were relatively new at the start of the project as well and as such there would have been a lot of overhead in writing any code at the start and probably even at my current experience point. On top of this Parse does not really support tests for databases as to do so you have to create a new Parse project.

In place of test driven development I opted instead for manual testing early on and once feature complete, exploratory testing. For manual testing I created a list of actions and expected behaviour. The tests are by no means comprehensive and test mostly the flow of the application rather than its state, hence the use of exploratory testing. Exploratory testing was conducted by users who had not seen the application in that state before. The benefits of this is that the users have no preconception of what is supposed to happen and as such do whatever feels natural. This can lead to the discovery of a lot more bugs than if the developer were to just write tests for what they think a user might do. Another benefit is that there is less overhead in setting it up. The application is simply given to a user and the developer can record user flows, bugs, and misunderstandings in the user experience.

The application has only been tested on the iPad Mini 4 as this was only physical device available during development. While Xcode does provide simulators to test applications on, it has limited support for location, only allowing a choice of several places, and no support for camera usage. Due to this it is not possible to test the application on the simulators.

Development Methodology

The project has been managed using a hybrid of Kanban and Agile. Agile is usually recommended for teams of at least four but it can still be a useful method of development for an individual developer (DXM, 2012). I decided to go with it as the methodology fits well with the weekly/fortnightly supervisor meetings.

Each sprint lasted two weeks and at the end of each sprint there would be a new working build that could be demonstrated during meetings. Feedback from these meetings could then be used to drive or alter the next sprint ensuring that the project did not go off track.

To help keep a stable version of the application that could be demonstrated, Git was used throughout development. The master branch would be the stable branch and new features would be worked on in feature branches. The use of Git branches allows developers to work on multiple features concurrently without having to worry about half implemented features. The benefit of this is that if I had hit a block on a certain feature I was free to move to another branch to work on something else. This prevented work from halting while trying to solve problems.

To keep track of progress I have made use of Trello. Trello is a web application that uses the kanban approach to manage projects. A Trello board has user defined lists which contain user defined cards. These cards are the tasks that need to be completed. The Trello board was ordered in a basic agile manner with 3 lists - backlog, current sprint and completed. After coming up with the scope and timeline of the project, the tasks were written to cards and divided into sprints, all of which were stored in the backlog. At the start of each sprint, the relevant cards would be transferred to the current sprint list and work would begin implementing them. Once a card had been implemented and tested, it was moved from the feature branch into the master branch ready to be demonstrated.

Implementation

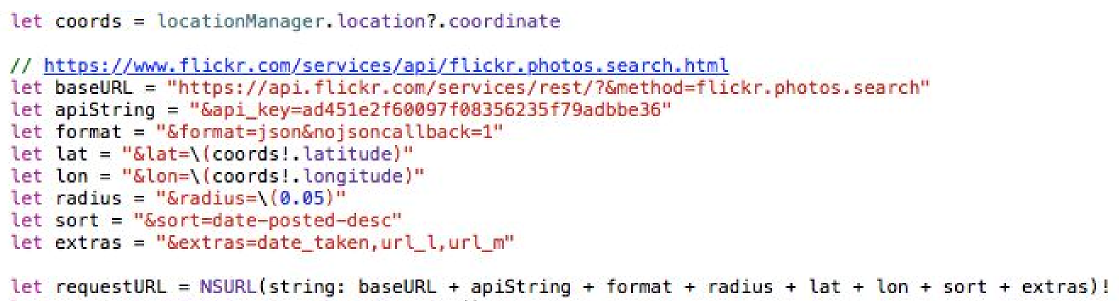
Database

The free level of Parse has a limit of 30 requests per second. This is measured minute by minute so the application is able to make 1800 requests per minute before getting cut off. The upper request limit can expanded up to 600 requests per second, 36,000 per minute, at the cost of $5,700.00. During the course of the project it is incredibly unlikely that the application will exceed the free tier, however scaling has been kept in mind. Due to this, the number of requests has been kept to a minimum by using techniques such as eager loading. Instead of fetching the post and then the creator, the creator is included in the initial request using `includeKey`:

```
let query = PFQuery(className: "Post")
query.includeKey("creator")
query.getObjectInBackgroundWithId(postId) {
    (post: PFObject?, error: NSError?) -> Void in
    ...
}
```

With regards to Flickr, the original plan was to use their `flickr.photos.getWithGeoData` API to retrieve photos around around the users location. However, this requires user authentication which is not preferable as users may not have Flickr accounts and either would have to create an account or not be able to view Flickr photos. After some more searching the `flickr.photos.search` API was found which does all of what `getWithGeoData` does including extras as well as not requiring user authentication.

The request URL is constructed by the following:



```
let coords = locationManager.location?.coordinate

// https://www.flickr.com/services/api/flickr.photos.search.html
let baseURL = "https://api.flickr.com/services/rest/?&method=flickr.photos.search"
let apiString = "&api_key=ad451e2f60097f08356235f79adbbe36"
let format = "&format=json&nojsoncallback=1"
let lat = "&lat=\(coords!.latitude)"
let lon = "&lon=\(coords!.longitude)"
let radius = "&radius=\(0.05)"
let sort = "&sort=date-posted-desc"
let extras = "&extras=date_taken,url_l,url_m"

let requestURL = NSURL(string: baseURL + apiString + format + radius + lat + lon + sort + extras)!
```

Flickr supports multiple response types including XML, PHP, SOAP, and JSON. The format requested for the application is JSON, with a standard response looking like this:

```
{
    datetaken = "2015-05-30 07:15:28";
    datetakengranularity = 0;
    datetakenunknown = 0;
    farm = 9;
    "height_l" = 683;
    "height_m" = 333;
    id = 17655772084;
    isfamily = 0;
    isfriend = 0;
    ispublic = 1;
    owner = "93509856@N05";
    secret = 361a1dc3ae;
    server = 8778;
    title = DSC08951;
    "url_l" =
"https://farm9.staticflickr.com/8778/17655772084_361a1dc3ae_b.
jpg";
    "url_m" =
"https://farm9.staticflickr.com/8778/17655772084_361a1dc3ae.jp
g";
    "width_l" = 1024;
    "width_m" = 500;
}
```

The reason for choosing JSON as the response type is because it is more widely used in API responses. The benefit of this is that if the application were to include sources from other image hosting platforms then it would be easier to make a generic class to parse the JSON. Another reason for selecting JSON is that it, in general, is shorter and requires less resources (Nurseitov, N., Paulson M., Reynolds R., Izurieta C., 2009). There is less boilerplate code around the values compared to say XML and as such requires a smaller payload. Similarly, the format is easier for developers to read which has its benefits when developing and debugging.

The radius has been kept to the same value as that used when querying the Parse database. This is to create a uniform service across all datasets. Similarly, the Flickr dataset is ordered in reverse chronological order like the parse dataset.

The Flickr API allows for extra information to be added to the query. In this case `date_taken`, `url_l`, and `url_m` have been included. `date_taken` is required because although Flickr sorts the photos for you, it simply uses the date and does not return it in the JSON. The is `ulr_l` is the URL required to retrieve the actual photo as, much like Parse, it is not returned with the JSON for bandwidth reasons.

Occasionally there is no `url_l` in some returned responses and as such would display only the placeholder image. Due to this, such items are filtered out on the application side using a closure:

```
flickrPhotosForLocation = flickrPhotosForLocation.filter() {
$0.valueForKey!("url_m") !== nil }
```

This iterates over the `flickrPhotosForLocation` array and removes items where `url_m` is non existent and returns the new array. `$0` is a shorthand argument name indicating the first argument.

Mobile

All of the view controllers with the exception of FullScreenViewController and SettingsViewController extend the ViewControllerParent class. This class contains common functionality that is used in many of the classes such as alerts, getting the current location and human friendly date conversion.

Activity indicators are used throughout the application to let the user know that something is going on in the background and that the application has not just simply stopped responding. The starting and stopping of this is also handled in the ViewControllerParent. It would have been preferable to include the activity indicator view inside of the parent class as well instead of having to create one for each view that used it but this was not possible as the parent class did not know which controller the indicator belong to. To get around this the indicator is defined in the relevant view controller and passed into the function in the parent.

```
func displaySpinner(spinner: UIActivityIndicatorView) {
    spinner.center = self.view.center
    spinner.activityIndicatorViewStyle = .Gray
    spinner.hidesWhenStopped = true
    view.addSubview(spinner)
    spinner.startAnimating()
}
```

This has lead to some repeated code but there is a benefit to this method of doing it. It allows for the creation of multiple activity spinners in the same view. For example, in the ViewPhotoViewController, it would be possible (although not currently used) to create individual activity indicators for the photo and the comments as they are brought in separately.

One issue encountered was the ordering of posts. While you are able to request the order of the returned results in the Parse query, it does not come back with the actual PFFile but rather a URL to retrieve it. Using these URLs the actual photo is then retrieved using `getDataInBackgroundWithBlock`. The issue with this is that there is no guarantee of return order so when the table is updated the images would not be in the correct order. To get around this problem the following struct was used:

```
struct Post {
    var postInformation: PFObject
    var photo: UIImage
}
```

So, instead of creating an array of `[PFObject]`, an array of `[Post]` is used to store the data used by the table. The `postInformation` contains the data returned from the initial query and the photo is the one eventually returned. This ensures that the photos in the table are in the intended order.

This technique is also used when retrieving the Flickr data set as it behaves much in the same way with the initial response containing a URL to the image and not the actual data. A dummy PFObject is created using the initial response data. This is by no means the best way to do it and the Post struct should be changed to have native swift types containing the relevant information.

Another issue encountered was refreshing the UITableView after retrieving the Flickr photos. When retrieving the photos from the Parse database it was possible to update the UITableView after each one so that each one would appear as they came in, however this did not work for the Flickr photos despite retrieving them in a very similar way. The solution appears to be updating the table on the main thread, like so:

```
// This way loads in the images as soon as they come in
// instead of all at once.
dispatch_async(dispatch_get_main_queue(), {
    self.imageCollectionView.reloadData()
})
```

This prevents the photos from appearing all at once, however issue is that sometimes when updating the table, the application drops user tapps making the application unresponsive. The issue does not happen all the time and as such this implementation has been left in, however I am not entirely happy with it and it should be reworked in the future.

As mentioned above, the post information is retrieved before the actual photo and as such the number of photos that are expected is know. To prevent the photos from ‘popping in’ and moving content unexpectedly around a placeholder image is put in as the photo until the actual one has been loaded. The same image is also used if the actual image is unable to be loaded to prevent crashing due to nil.

Since the application photos are retrieved before the Flickr photos, the two sources have been split into separate sections. This means that the `[Post]` array does not have to be sorted. From a user experience point of view it also prevents elements shifting unexpectedly when loading the Flickr data set.

One thing kept in mind during development was that the user would be using the application while on their mobile data. Since the home screen is the one that contains the most images, thumbnails were used to decrease the amount of data required to load it. To create these, when the user shares a photo, the device also creates a thumbnail version:

```
func createThumbnail(image: UIImage) -> UIImage {
    // Scales the image to 10% of current size.
    let size = CGSizeApplyAffineTransform(
        image.size,
        CGAffineTransformMakeScale(0.1, 0.1))
    let hasAlpha = false
    let scale: CGFloat = 0.0

    // Setting hasAlpha to false for images without transparency
    // may result in a pink hue.
    UIGraphicsBeginImageContextWithOptions(size, !hasAlpha, scale)
    image.drawInRect(CGRect(origin: CGPointZero, size: size))

    let scaledImage = UIGraphicsGetImageFromCurrentImageContext()
    UIGraphicsEndImageContext()

    return scaledImage
}
```

This newly created thumbnail is then compressed to the same level as the full size image then stored with the post to be used in the home screen photo previews and also on the view photo screen while waiting for the full sized one to be retrieved.

The performance benefits of using thumbnails are mentioned in the Results and Evaluation section.

One of the features mentioned in the project scope has not been implemented is uploading images at a later time if the user does not have network connection at the current time. The initial plan was to use the `saveEventually` which, given no network connectivity, writes the data to local storage and attempts to resend this at a later date. I have used this in previous Android projects but was unable to get it to work using iOS and Swift. The main difficulty came from the lack of any feedback from Parse. There were no error messages mentioning that something had gone wrong yet the object was never saved to the Parse database. There were many suggestions as to fixing this problem ranging from restarting the application to waiting for over 10 minutes but none of these worked. Due to lack of time and the amount of time between making a change and testing to see if would work, the feature was dropped for the project.

A method considered of implementing the above functionality without Parse would be to check for network connection, if there is none then save it to `NSUserDefualts`. After a given amount of time or when the application has been restarted, the application could check for network connectivity and also check that the `NSUserDefualts` storage was not empty and attempt to resend again. However, due to lack of time this was not attempted.

The user interface is implemented using Storyboards. This not only allows the developer to place interface elements but allows for screen transition information as well as setting up delegates and data sources. The benefit of this is that it removes some code from the view controllers, however, it adds another place to look for functionality when changes are being made. Another downside of using the interface builder rather than code is coupling. The view is very tightly coupled to the view controller which makes reuse of elements difficult if not impossible. So, if I were, for example, to allow for commenting on a location, and not just a photo then the interface would have to be remade instead of simply reusing the comments section.

The reason for choosing storyboards over code was for rapid development. It is very quick to create a storyboard that will scale on all devices and link it to code. Most, if not all, of the options of an interface item are shown to the developer which helps them quickly find what is needed, for example, setting the keyboard to all capital letters which is used in tagging.

Auto layout was also used when creating the user interface. This allows the developer to define relationships between other elements or the screen edges. For example, an element can be pinned to the bottom centre of the screen so no matter what screen dimensions a user is using, that element will always be at the bottom centre.

Web

The administrator web interface is implemented using the Model-View-Controller architecture.

The views have been split into fragments to allow for more code reuse and to prevent duplication, for example, there is a `master.blade.php` that yields content with other views, such as `admin.index.blade.php` provide. The `master.blade.php` file contains the header as well as CSS and JavaScript imports that will be needed throughout the project such as JQuery. The exception to this is the login page. This has been kept separate as there are more elements that would have to be hidden in the `master.blade.php` file than would be duplicated.

A security feature that is provided by Blade is XSS protection. By enclosing outputted user generated data in triple curly braces it will do the equivalent of calling `htmlentiies()` in basic PHP. Example, if a user were to create the comment `<script>window.alert("XSS Attempt");</script>` and then report it, running it through the code `{{{ $comment->get('comment') }}}` will prevent the code from being executed in the browser.

Another form of security that has been included is the use of routing group filters. These allow you to perform checks before or after a route is used. The filter created is called `auth` which allows the web application to check to see if a user is authenticated before allowing a user to load the page.

```
Route::filter('auth', function() {
     if (ParseUser::getCurrentUser() == null) {
          return Redirect::route('login');
     }
});
```

In order to use this filter, an anonymous function must be created, containing all the routes that requires filtering, for example:

```
Route::group(['before' => 'auth'], function () {
    Route::get('/', array(
        'as' => 'home',
        'uses' => 'AdminController@index'
    ));
```

```
     Route::get('logout', function() {
        ParseUser::logOut();
        return Redirect::route('login');
    });
     ...
     ...

});
```

To prevent constant refreshing when an administrator decides to allow or delete a reported item, they are removed using AJAX. The code for delete and allow are largely the same. They both make use of a data-id attribute that is added to each row in the table when generated. The delete also request also makes use of a data-class attribute as it not only needs to delete the item from the report table but also from the original table. Once the item has been dealt with the row is then removed from the table.

```
function deleteItem(element) {
  $.ajax({
    type: "POST",
    url: "{{ route('delete-item') }}",
    data: {
      objectId: element.getAttribute("data-id"),
      objectClass: element.getAttribute("data-class")
    }
  }).done(function(msg) {
    element.closest("tr").remove();
  });
}
```

Results and Evaluation

Experiments

One thing kept in mind throughout the development of the application was that often the user would be on their mobile data and as such the application should minimize the usage as much as possible without negatively impacting the user experience. Due to this a number of experiments were conducted in order to see how the application could reduce data usage.

The first experiment was with the levels of compression. When converting from UIImage to data, iOS allows the developer to specify the level of JPEG compression they want. The goal was to allow for the most compression without a perceivable drop in image quality. The compression quality levels used in these experiments were 1.0, 0.5, and 0.3. Using the same photo setup the size on disk in megabytes for the images were as followed:

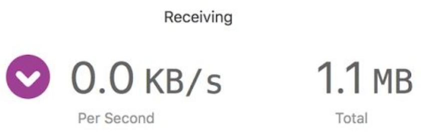
- Quality 1.0: 5.70 MB
- Quality 0.5: 0.97 MB
- Quality 0.3: 0.50 MB

Although there was little difference in quality when looking at the photos in full, when the zoomed in to 500%, compression quality 0.3 started to show unacceptable levels of compression artifacts and as such 0.5 was selected to be used in the application.

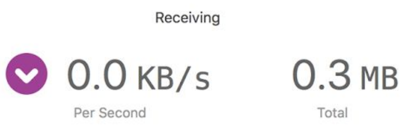
Outside of just performance benefits for users, these levels of compression allow for more photos to be stored in the Parse database. Parse, at the lowest tier allows for 20 GB of images. At the original photo quality, there would only be room for approximately 3,500 photos but at a compression quality of 0.5 there is enough room for 20,000 photos.

The second experiment conducted was making use of thumbnails to reduce the data usage from on the home screen. Again, the aim was to minimize the data usage while not creating a noticeable drop in photo quality. Using 4 photos shared from within the application and 3 from Flickr, the application required the following network usage to load the home page:

- No thumbnails: 1.1 MB
- Thumbnails: 0.3 MB

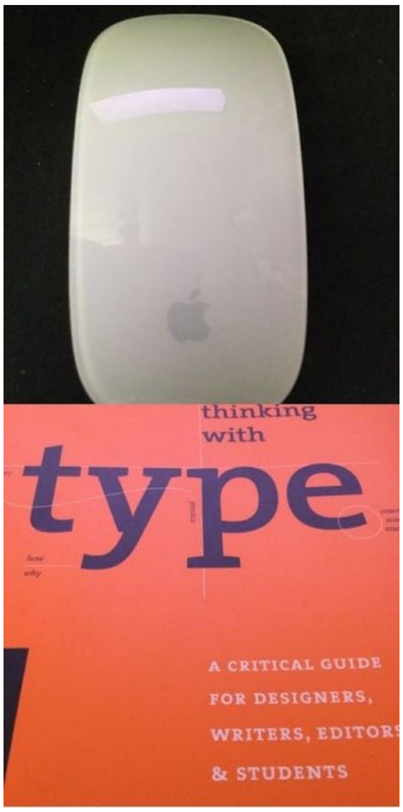


HomeViewController network usage before thumbnails

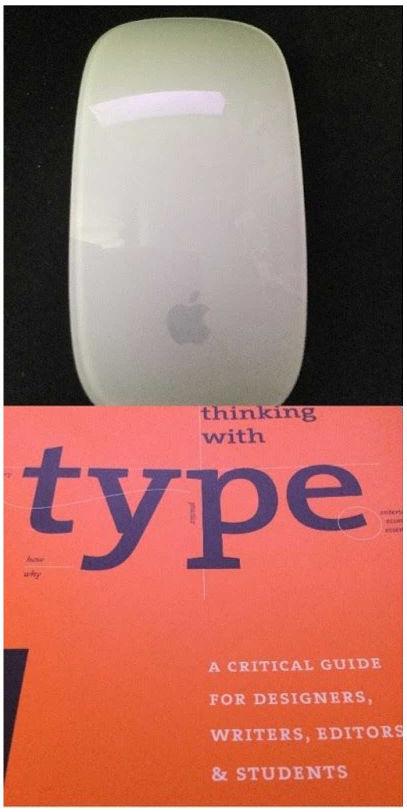


HomeViewController network usage after thumbnails

One thing that should be noted it that there appears to be some compression done automatically by iOS or Parse as 1 photo takes up ~1 MB on disk but with 7 photos there is only 1.1 MB of network usage. Even with this, the use of thumbnails has reduce the network usage to under 30% of the original which I would deem a success.

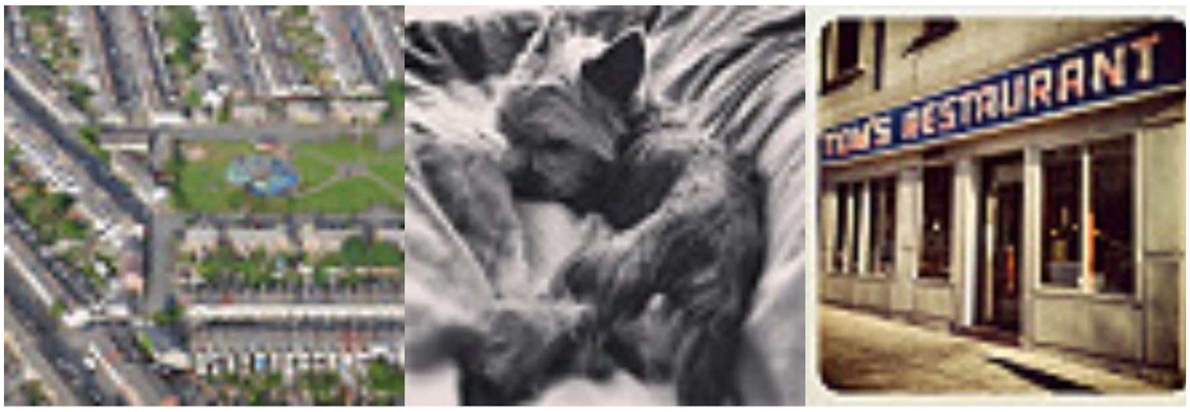


Sample of HomeViewController photos before thumbnails

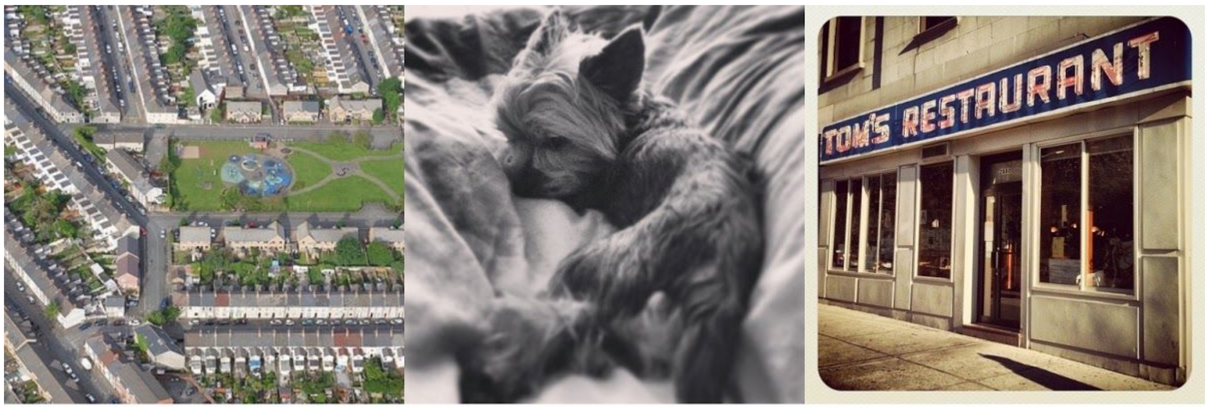


Sample of HomeViewController photos after thumbnails

Although I am unable to control the level of compression for photos that come from Flickr, the API does provide a number of images sizes such as url_t (thumbnail), url_m (medium), and url_l (large). I first attempted to use url_t but found the image quality dropped to below acceptable levels so opted instead for url_m.



Sample of Flickr photos using url_t



Sample of Flickr photos using url_m

As you can see, there is significant difference between the two qualities.

User Tests

User testing was carried out via exploratory testing. The user was told the purpose of the application then given the device with the application running, starting on the login view, and given no direction from the observer. Questions asked by the user were recorded but no answered during the test to try to keep the new user experience as realistic as possible. All of the users for this section had not seen the application before.

General results are as follows:

LoginViewController - As no users had accounts, this was not particularly tested. All users did, however, quickly find the button to take them to the signup page.

SignUpViewController - No users had difficulty creating an account for themselves, however, after filling in the final field the majority of them attempted to submit the information by tapping the return button on the keyboard. In the future it may be worthwhile to link the keyboard return button to the submit button to create less work for the user.

HomeViewController - In general there seemed to be confusion as to the criteria for both the range photos were used from as well as the sorting order. There was also some confusion about why recently shared photos were not on the home screen. All but one of the users tried pull down to refresh but some work should be done to find a way of adding user shared photos to the home screen as they are shared.

The issue with loading Flickr photos on the main thread causing unresponsiveness from the application occurred with one user making them have to tap the photo they wanted to see multiple times before it would load.

SharePhotoViewController - Users had no trouble with photos and comments, however, the majority of the users typed a tag but did not tap the ‘Add Tag’ button to actually add it to the post. Two users attempted to click on the photo they had just taken trying to see it in fullscreen.

One user accidently tapped ‘Cancel’ when going to submit which instantly cleared the screen. An alert should be added to confirm that it’s the action that the user wishes to do.

ViewPhotoViewController - All but one users were unable to find the report feature. The one user that did find it found it by accident and only on the photo, not the comments. This confirms the worry that users would have difficulty finding the feature.

Several users tried swiping left and right in an attempt to go to the next photo. This is something I had not considered doing but it seems like a feature worth implementing.

SettingsViewController - There were no issue on this screen although one user commented that the elements should have more space between them.

ManagePostsViewController - The only time users entered this was before they had shared any photos. In such a case it may be worth putting in a call to action, something along the lines of ‘You haven’t shared any photos yet’ with a button taking them to the SharePhotoViewController.

General observations - Overall navigation was not an issue with users being able to find what they were looking for reasonably quickly. One thing that should be improved however is feedback for certain states, especially empty ones like that found in the ManagePostsViewController when no photos have been shared. This would prevent users from having to look at a blank page and wondering what it was for or if it was still loading.

Manual Tests

LoginViewController

Description	Action	Expected Outcome	Pass/Fail	Notes
Attempt login with incorrect details.	1. Username: ‘Chris’ 2. Password: ‘p’ 3. Tap ‘Log in’	● Error: ‘Invalid login parameters’ ● Fields still contain user inputs ● End on LoginViewController	Pass	‘Invalid’ in error message does not begin with a capital letter.
Attempt login with no username.	1. Username: nil 2. Password: ‘a’ 3. Tap ‘Log in’	● Error: ‘You must fill in all fields’ ● Fields still contain user inputs ● End on LoginViewController	Pass	
Attempt login with no password.	1. Username: ‘a’ 2. Password: nil 3. Tap ‘Log in’	● Error: ‘You must fill in all fields’ ● Fields still contain user inputs ● End on LoginViewController	Pass	
Attempt login with no username or password.	1. Username: nil 2. Password: nil 3. Tap ‘Log in’	● Error: ‘You must fill in all fields’ ● Fields still contain user inputs ● End on LoginViewController	Pass	
Attempt to login with correct details.	1. Username: ‘Chris’ 2. Password: ‘a’ 3. Tap ‘Log in’	● Activity indicator appears while waiting for login ● User logged in as ‘Chris’ ● Redirect to HomeViewController	Pass	
Go to SignUpViewController.	1. Tap ‘Sign Up’	● Redirect to SignUpViewController	Pass	

SignUpViewController

Description	Action	Expected Outcome	Pass/Fail	Notes
Attempt signup with details that already exist.	1. Username: 'Chris' 2. Password: 'a' 3. Retype password: 'a' 4. Tap 'Sign Up'	● Error: 'Username Chris already taken.' ● Fields still contain user inputs ● End on SignUpViewController	Pass	'Username' in error message does not begin with a capital letter.
Attempt signup without matching passwords.	1. Username: 'a' 2. Password: 'a' 3. Retype password: 'z' 4. Tap 'Sign Up'	● Error: 'Passwords do not match.' ● Fields still contain user inputs ● End on SignUpViewController	Pass	
Attempt signup without username.	1. Username: nil 2. Password: 'a' 3. Retype password: 'a' 4. Tap 'Sign Up'	● Error: 'You must fill in all fields.' ● Fields still contain user inputs ● End on SignUpViewController	Pass	
Attempt signup with all fields empty.	1. Username: nil 2. Password: 'a' 3. Retype password: 'a' 4. Tap 'Sign Up'	● Error: 'You must fill in all fields.' ● Fields still contain user inputs ● End on SignUpViewController	Pass	
Attempt sign up with correct details.	1. Find username that does not exist 2. Password: 'a' 3. Retype password 'a' 4. Tap 'Sign Up'	● Activity indicator appears while waiting for signup ● Redirect to HomeViewController ● User logged in as newly created user.	Pass	
Go to LoginViewController.	1. Tap 'Log in'	● Redirect to LoginViewController	Pass	Black screen flashes during transition.

HomeViewController

Description	Action	Expected Outcome	Pass/Fail	Notes
Pull down to refresh	1. Pull down from near top of view	● Spinner to show activity directly after pulling down ● After refresh: Same photos as before in the same order	Pass	Although it does refresh, the view can update before the user lifts their finger.
View Application Photo	1. Tap photo under ‘Geo Snap’ table heading	● Redirect to ViewPhotoController ● Photo is same as one tapped	Pass	
View Flickr Photo	1. Tap photo under ‘Flickr’ table heading	● Redirect to ViewPhotoController ● Photo is same as one tapped	Pass	Takes longer to go to ViewPhotoController than tapping a photo under ‘Geo Snap’.
Go to SharePhotoViewController	1. Tap ‘Take Photo’	● Redirect to SharePhotoViewController	Pass	
Go to SettingsViewController	1. Tap ‘Settings’	● Redirect to SettingsViewController	Pass	

ShareViewController

Description	Action	Expected Outcome	Pass/Fail	Notes
Attempt to share without taking photo.	1. Tap ‘Share’	● Error: ‘You must take a photo to share’	Pass	
Take photo.	1. Tap ‘Take Photo’	● iOS camera application launched ● Photo taken replaces placeholder image	Pass	
Attempt to share after taking photo but with no comment or tag.	1. Take photo 2. Tap ‘Share’	● Activity indicator appears while upload takes place ● Redirect to ViewPhotoViewController ● Display username, time, and date ● Photo is the one taken	Pass	
Add a tag.	1. Type ‘TEST’ into tags input 2. Tap ‘Add Tag’	● ‘TEST’ should be displayed below input	Pass	
Remove a tag.	1. Type ‘TEST’ into tags input 2. Tap ‘Add Tag’ 3. Tap the ‘TEST’ tag	● Tag should be removed from below the tag input	Pass	
Attempt to share after taking photo, adding comment and tags.	1. Take photo 2. Add comment ‘This is a test comment’	● Redirected to ViewPhotoViewController ● Display username, time, and	Pass	

	3. Add Tag ‘TEST’ 4. Add Tag ‘HELLO’ 5. Tap ‘Share’	date ● Photo is the one taken ● Comment ‘This is a test comment’ below photo ● Tags ‘TEST | HELLO’ below comment ● Refreshing HomeViewController displays newly shared photo		
Cancel share.	1. Take Photo 2. Add comment ‘test’ 3. Add tag ‘test’ 4. Tap ‘Cancel’	● Photo should be removed. ● Comment should be removed. ● Tags should be removed.	Pass	
Go to HomeViewController	1. Tap ‘Home’	● Redirect to HomeViewController	Pass	
Go to SettingsViewController	1. Tap ‘Settings’	● Redirect to SettingsViewController	Pass	

ViewPhotoViewController

Description	Action	Expected Outcome	Pass/Fail	Notes
View fullscreen	1. Tap on photo	● Redirect to FullScreenViewController ● Photo should be one from previous screen	Pass	
Like photo	1. Tap on the heart outline icon	● Heart icon should become filled ● Leaving screen and coming back should maintain liked status.	Pass	
Unlike photo	1. Tap on the full heart icon	● Heart icon should become an outline ● Leaving screen and coming back should maintain non-liked status	Pass	
Comment on photo	1. Tap in ‘Add Comment’ input 2. Add comment ‘Test comment’ 3. Tap ‘Submit’	● Comment ‘Test comment’ should be added to the top of the comments list	Pass	
Report photo	1. Longpress on a ‘Geo Snap’ photo 2. Tap ‘OK’	● Alert: ‘Report Successful - Your report has successfully been sent.’	Pass	
Report comment	1. Longpress on comment	● Alert: ‘Report Successful - Your report has successfully been	Pass	

	2. Tap ‘OK’	sent.’		
Close screen	1. Tap ‘Close’	● Redirect to previous view	Pass	

FullScreenViewController

Description	Action	Expected Outcome	Pass/Fail	Notes
Zoom	1. Spread fingers on photo	● Photo should be expanded ● User should be able to scroll around photo	Pass	
Close	1. Tap ‘Close’	● Redirect to previous view	Pass	

SettingsViewController

Description	Action	Expected Outcome	Pass/Fail	Notes
Go to ManagePostsViewController	1. Tap ‘Manage Posts’	● Redirect to ManagePostsViewController	Pass	
Go to HomeViewController	2. Tap ‘Home’	● Redirect to HomeViewController	Pass	
Go to SharePhotoViewController	2. Tap ‘Take Photo’	● Redirect to SharePhotoViewController	Pass	
Log out	1. Tap ‘Log Out’	● Redirect to	Pass	

		LoginViewController		

ManagePostsViewController

Description	Action	Expected Outcome	Pass/Fail	Notes
Delete post	1. Slide post left 2. Tap ‘Delete’ 3. Tap ‘Delete’	● Post should be removed from the table view ● Refreshing HomeViewController should remove the photo (if at that location)	Pass	
Close	1. Tap ‘Close’	● Redirect to SettingsViewController	Pass	

Conclusion

The goal of the project was to create an iOS application that allowed users to share photos that are linked to a location. Other users could then interact socially with these photos. The ability to share photos later if there is currently no network connection has not been implemented but apart from that I have met all the aims of the project and I am happy with the current state of the application. An overarching aim was to create a product that would be ready to release and barring the lack of a commercial Flickr API key, I believe it is at that point.

I am happy with how development progressed, however I should have spent more time learning about and implementing automated testing. Using the agile methodology was very helpful in managing progression and the use of Git to maintain a demo branch worked out well as I was often working on a feature when the application needed to be demonstrated.

One of reasons for choosing this project to work on was to learn about iOS development and I believe I have come a long way. I would like to work further on this project in the future and use it as a vehicle to apply new things I learn about such as testing and user interface design. Working on this project has only increased my interest in iOS development.

Future Work

Database

As of now, the database is hosted using Parse. However, on January 28th, 2016 it was announced that Parse will be shutting down on January 28th, 2017 (Parse, 2016, Moving on). Due to this it will be necessary to migrate the database to another service. Luckily Parse have made ParseServer open source and have a guide on migrating an existing application to be self-hosted (Parse, 2016, Parse Server Guide). This would be one of the first steps to take due to time sensitivity. It will also be easier doing it as early as possible due to the lower number of users.

If the application were to move away from Parse in general, it would be necessary to create pivot tables to replace the Parse pointers that have been used for convenience.

While migrating the database it would also be beneficial to make field and table names consistent. Due to expanding the database as needed, there are some inconsistencies such as the description of a photo being called a comment. Although this is technically correct, this may lead to some confusions. Now that I am more aware of the data required, it would be possible to redesign areas of the database such as creating an administrator table instead of including an isAdmin column in the users table. Similarly, on the mobile interface, comments that are too long are simply cut off. It would be worthwhile to limit the character count in the database to prevent excess information being stored. The users could then be notified of this limit.

Creating an audit table that tracked the actions of administrators would, at some point, be necessary. It would need to track which administrator has done what action to what post. For this to be effective, soft deletions would also have to be implemented.

The Flickr API is for non-commercial use. To use it for commercial use you must apply for a commercial API key. So, while I believe I have met the aims for the project technically, there are still some steps that would need to be taken to allow the application to be released.

Mobile

While not a high priority for the user, it would be worthwhile to spend some time refactoring the codebase. A lot has been learned about both Swift and iOS development over the course of the project and as such there are many places the code could be improved. One such area would be to use interfaces to decouple the application from the backend. Similarly, the majority of the code is in the view controllers which mixes up responsibilities making expanding the project more and more difficult as it gets larger. The code should be refactored into a more object oriented approach and possibly switch to an architecture similar to MVC.

One feature that has not been implemented was the ability for the application to save a photo later if there is no network connection. The plan was to make use of Parse's save eventually method which saves objects to local storage until the device gains network connection then attempts to save the object. However, I was unable to get this to work. I still believe this to be a useful feature for users and as such should be looked into if work were to continue on the project.

The user interfaces are very basic. For example, if a comment is too long then the excess is simply cut off. This should be addressed by either expanding the labels to fit the entire comment, allowing users to tap the comment to view the full thing, or by enforcing a character limit. The interface should also become more responsive to screen orientation. For the most part everything works when the screen is in landscape, however, there are some usability issues such as only being able to view one comment at a time on the ViewPhotoViewController. To address this there should be custom layouts for certain situations. For example, if the device is in landscape on the ViewPhotoViewController then the post information (photo, creator, tags, description) should move to the left side and the comments could be on the right side. A similar approach would appropriate for the SharePhotoViewController.

Another issue with the user interface is the discoverability of the report function. As mentioned above, only one user managed to find it and that was by accident. As of now, I am unable to think of a better approach but it is certainly something that would need to be reworked. One approach would be to provide new users with a 'tour' of the application to show them what is possible.

At the moment the application allows users to like photos. The number of likes a post has should be displayed, either on the HomeViewController or the ViewPhotoViewController. This can act as positive reinforcement for the content creators as well as can indicate quality posts to the consuming user.

New features to work on would include multiple methods of sorting content the HomeViewController. Possibilities could include number of likes, not just overall but possibly a velocity of likes over time or some type of diminishing return system where the first 10 likes is equivalent to the next 100 or such.

The next feature that I believe would add the most value to the user as well as potentially increasing the number of people who would be interested in using the application would be to create recommendations for nearby places. The initial idea for this is to use tags and likes to find places that have similar tags and preferably a lot of likes from users. By looking at the tags of a user's most liked posts it should be possible to find locations nearby that the user may like to visit. For example, if a user has liked 3 photos with the tag 'HISTORY', we could look around the area for places that the user has not liked, has the tag 'HISTORY' and preferably a large number of likes.

Web

As mentioned in the approach section, moderation is not scalable. If the need for multiple moderators were to arise then it would be worth researching socket use to allocate work real time. One approach I have considered would be to display one item on the screen at a time and once that has been dealt with the server could decide on the next item to be looked at. Using sockets, the server could create a buffer of items to be looked at for each open socket ensuring no work is repeated. However, due to the low number of users this feature would be a very low priority.

In the future it may also be preferable to allow administrators to ban problematic users. One way I have thought of doing this is count how many items a user has created has been reported, if it exceeds a threshold then an administrator could be notified to decide on if the user should be ban or not. To make this more robust it would be worth storing the device IDs an account has been used on in a pivot table to prevent ban users from creating new accounts.

A feature of medium priority would be to ensure that non-administrators are not able to create a POST request to the API to delete or allow items. Realistically it would be difficult for them to get access to any objectIds without having access to the administrator pages which would require them to be authenticated but just in case it were possible, it would be worthwhile to create safeguards against this.

General

One area neglected throughout development is automated testing. With developing as well as learning Swift and iOS, it was hard to find time to learn about and implement automated testing but now that I am more comfortable with these the next thing to work on is testing. Most, if not all of the manual tests can be converted to automated testing. The majority of the codebase should have tests written for them and for future development TDD would be preferable. The current test document should be implemented by automated testing. Furthermore, including continuous integration would be worthwhile, especially if there were to be more developers working on the project.

Reflection on Learning

A fair portion of the codebase is not as efficient as it could be. This is partly due to lack of experience both with Swift and iOS as well as trying to keep up with the project plan. A number of times during development I would implement something that would work but I was not entirely happy with such as how liking and unliking are managed by insertion and deletion of rows. The plan was to go back and create a better solution but due to the nature of the project, there was always something new to be working on that would create more value to the user. There is something to be said for not optimising too early but sometimes solutions that work are not suitable and should be changed early on to prevent technical debt from building up. The effects of this have been felt even in this relatively short project and so in larger/longer projects it is really imperative to keep on top of the codebase to prevent it from getting too much out of hand.

A few times during the project after implementing a new feature there was some regression. One such example was adding Flickr integration. Changes made to the HomeViewController lead to breaking the indexing of the collection view causing the application to crash when tapping a GeoSnap photo. These such bugs were picked up and fixed quickly, however as the project grows it may become harder to find these. This really highlights the need for automated testing as although they probably took less time to identify and fix than learning about and implementing tests, tests are an integral part of software development and time should be allocated to implementing them to cover the codebase and protect against regression.

At the start of the project, I researched similar applications and drew some inspiration with regards to the user interface. A lot of the applications would separate the comments from the photos. For the project application I decided to put the comments and the photos on the same screen because they were of similar value to the user. However, after creating the ViewPhotoViewController view I have come across some interesting problems such as when scrolling comments, should the whole view move up hiding the photo or should just the comments move? When encountering these problems it creates a better understanding for why other people have made certain decisions.

Working with a relatively new platform and language to create this project has helped improve confidence when learning ‘on the job’. One thing that helped was my experience with Android development. Knowing what I would need to do in Android helped guide my research when looking for help with iOS. For example, knowing about Android intents helped guide me to find delegates which was used when using the camera. Some areas, however, good practices from Android were not useful such as when developing the user interface. For the most part, using the interface builder in Android produces bad code so it is recommended to create it via XML, however, using storyboard and autolayout in iOS allows the creation of interfaces that scale on multiple screen sizes.

One thing I should have done towards the start of the project was look at the source code of other iOS projects that have been made. Good project examples provide a wealth of good practices as well as introduce you to new concepts that you might not be aware of such as class extensions and other uses of delegation. Extensions could have been used in the Parse class to convert the time to a more user friendly format instead of putting that code in the ViewControllerParent where it does not exactly belong. Having more knowledge of delegation early on would have given a solid starting point to decoupling views and controllers as it is used to pass information from one class to another. This could have been used to create a separate class to retrieve posts from Flickr or Parse.

References

Xamarin (2016). [online] Available at: https://www.xamarin.com/ [Accessed 26 April 2016].

TIOBE Index (2016). [online] Available at: http://www.tiobe.com/tiobe_index?page=index [Accessed 3 May 2016]

Cristian González García et al., (2015), Swift vs. Objective-C: A New Programming Language. [online] available at: http://www.ijimai.org/journal/sites/default/files/files/2015/05/ijimai20153_3_10_pdf_19818.pdf [Accessed 4 May 2016]

Flickr (2016). [online] Available at: https://www.flickr.com/ [Accessed 26 April 2016].

Current Location (2016). [online] Available at: http://current-location.com/ [Accessed 11 April 2016].

Piximity (2016). Redwheel Apps. [online] Available at: http://www.piximity.me/index.html [Accessed 11 April 2016].

Aku Kolu (2012), MVC Frameworks in Web Development, pp. 15-16. [online] Available at: https://jyx.jyu.fi/dspace/bitstream/handle/123456789/40636/Aku%20Kolu.pdf [Accessed 4 May 2016]

DXM (2016), How can a single developer make use of Agile Methods? [online] Available at: http://programmers.stackexchange.com/a/141821 [Accessed 4 May 2016]

Nurseitov, N., Paulson M., Reynolds R., Izurieta C. (2009), Comparison of JSON and XML Data Interchange Formats: A Case Study. [online] Available at: https://www.cs.montana.edu/izurieta/pubs/caine2009.pdf [Accessed 4 May 2016]

Parse (2016). Moving on. [online] Avilable at: http://blog.parse.com/announcements/moving-on/ [Accessed 18 April 2016].

Parse (2016). Parse Server Guide. [online] Available at: https://github.com/ParsePlatform/parse-server/wiki/Migrating-an-Existing-Parse-App [Accessed 18 April 2016].