```
//create image view and load the post image (and the rest of the data) in
    a separate thread for smooth scrolling
//set a image placeholder before the async call will be performed
UIImage *placeholderPhoto = [UIImage imageNamed:@"image-placeholder"];
[oneCell.postImage setImage:placeholderPhoto];

//run async block
dispatch_queue_t queue =
    dispatch_queue_create("com.photoCampaign.postImageQueue", NULL);
dispatch_async(queue, ^{
    //load the image from photocampaign.net
    NSString *urlString = [[NSString alloc] initWithFormat:@"%@%@%@",
        REQUEST_URL, IMAGE_BASE_SLUG, [currentPost photoURL]];
    NSURL *photoURL = [[NSURL alloc] initWithString:urlString];
    UIImage *loadedPostPhoto = [UIImage imageWithData:[NSData
        dataWithContentsOfURL:photoURL]];

    //update UI after image load
    dispatch_async(dispatch_get_main_queue(), ^{
        //update the UI on the main thread
        [oneCell.postImage setClipsToBounds:YES];
        [oneCell.postImage setImage:loadedPostPhoto];
        //set imageview to scale image to it's frame
        [oneCell.postImage setContentMode:UIViewContentModeScaleAspectFill
            ];

        //trigger display update
        [oneCell.postImage setNeedsDisplay];
        [oneCell.postImage setNeedsLayout];
    });
});

//update post author label
oneCell.postAuthor.text = [NSString stringWithFormat:@"%@ %@",
    [currentPost authorFirstName], [currentPost authorLastName]];

//load relative time of post
    //convert the JSON date first
    NSDateFormatter *dateFormatter = [[NSDateFormatter alloc] init];
    [dateFormatter setDateFormat:@"yyyy-MM-dd'T'HH:mm:ss.SSSZ"];
    NSDate *postCreatedDate = [dateFormatter dateFromString:[currentPost
        created]];
    //convert to relative post date
    NSString *relativePostDate = postCreatedDate.timeAgoSinceNow;
//update label
oneCell.postDate.text = relativePostDate;

//update title of the post
oneCell.postTitle.text = [currentPost title];

return oneCell;
```

```
/**
 * Create a post
 */
exports.create = function(req, res) {
  //read post data
  var postData = {},
  fileUrl = './public/uploads/',
  uploadMessage = '',
  isiOS = req.body.isiOS;

  if (!req.files || req.files.postPhoto.size === 0) {
    uploadMessage = 'No file uploaded at ' + new Date().toString();
    return res.send(400, {error:uploadMessage});
  } else {
    var file = req.files.postPhoto;
    //append filename and date to file upload
    fileUrl += new Date().getTime().toString() + file.name;

    fs.rename(file.path, fileUrl, function(err) {
      if(err) {
        return res.send({
          error: 'Error while moving the file: ' + err
        });
      } else {
        uploadMessage = '<b>"' + file.name + '"<b> uploaded to the server at ' + new Date().toString();

        if(!isiOS){
          //store data from req params
          postData.title = req.param('title');
          postData.description = req.param('description');
        }
        else{
          //handle iOS specific requests
          //store data from req params
          postData.title = req.body.title;
          postData.description = req.body.description;
        }
```

# Photo campaign platform report

Prepared for: Mobile Application elective final mandatory assignment
Prepared by: Dan Mindru
19 May 2014

Notice: the screenshots presented in this document represent the current state of the web & iOS application and may have been improved or extended in the meantime. (Differences may be visible on photocampaign.net)

PHOTO CAMPAIGN PLATFORM REPORT 1
INTRODUCTION 3
CLASSES AND STORYBOARD 4
OWN CLASSES 5
APPLICATION FLOW 7
TABLE VIEW IMPLEMENTATION 8
CELL INSTANTIATION 8
SORTING CELLS 9
PULL TO REFRESH 10
PHOTO EDITING IMPLEMENTATION 11
KEYBOARD HIDING AND RETURN KEY 12
PODS AND THIRD PARTY LIBRARIES 13
CONCLUSION 14

# INTRODUCTION

The following document will explain important parts of the code behind the iOS application of the Photo Campaign Platform.

The code is available on Github: github.com/dandaniel/photo-campaign-app

Warning: In order to build the application from the attached zip file, **photoCampaign.xcworkspace** needs to be opened in XCODE.

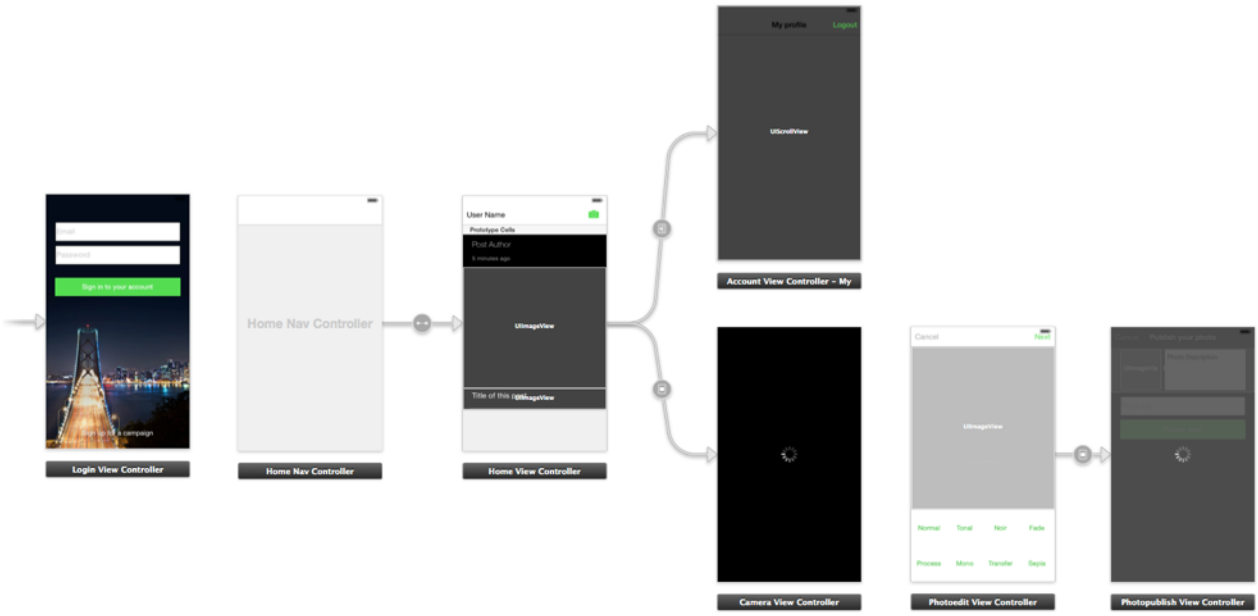
A **video** demo of the entire platform is available at youtube.com/watch?v=xlu2rq1LZaY

The video begins with the presentation of the web platform, then continues with describing technical information (code explanations for the web back-end and front-end). Following, a short iOS code explanation is provided and finally the iOS app is demoed in relation to the web platform.

It is recommended that the video is watched after reading this document.

# CLASSES AND STORYBOARD

The storyboard (mainStoryboard) represents the collection of all the view controllers (and child elements) of the application (no additional NIB files used).




There are a total of 7 controllers (one of which is a HomeNavController - Navigation View Controller in which the HomeViewController - Table View Controller is embedded) used in this app. Segues are done both using Interface Builder and programatically (for login or photo editing).

Each view is subclassed as follows (left to right):

The login view: **View Controller**
subclassed as **loginViewController (<UITextFieldDelegate>)**

The post feed view: **Table View Controller**
subclassed as **homeViewController (<UITableViewDelegate>, <UITableViewDataSource>)**

The account view (top): **View Controller**
subclassed as **accountViewController**

The camera view (bottom): **View Controller**
subclassed as **cameraViewController**

The photo editor (bottom): **Scroll View Controller**
subclassed as **photoeditViewController**

The post publisher (bottom): **View Controller**
subclassed as **photopublishViewController (<UITextFieldDelegate>)**

Other subclassing is done for the Table View in the homeViewController and it's children Table View Cells. Lastly, a UITextField is subclassed as 'accountTextField', which draws a input that has a bigger height and additional padding for the login and photo publishing view.

# OWN CLASSES

Without the subclasses above, there are 3 additional classes created: **user**, **post** and **statics**.

The **statics** class contains only the constant variables (mostly URL request data) that are used across the application.

```
#define REQUEST_URL @"http://photocampaign.net"
#define LOGIN_SLUG @"/auth/signin"
#define IMAGE_BASE_SLUG @"/"
#define POST_FEED @"/posts"
#define POST_CREATE_SLUG @"/posts"

@interface statics : NSObject

@end
```

The **user** and **post** are quite similar, both containing a custom initialisation method that sets all their properties to a passed value. They both have methods that allow saving instance data in a **plist**, as well as reading or deleting that data.

Additionally, the post class contains a 'imageWithImage:scaledToSize method, which is a helper method to resize photos before editing and before uploading to the server.

**post** class interface:

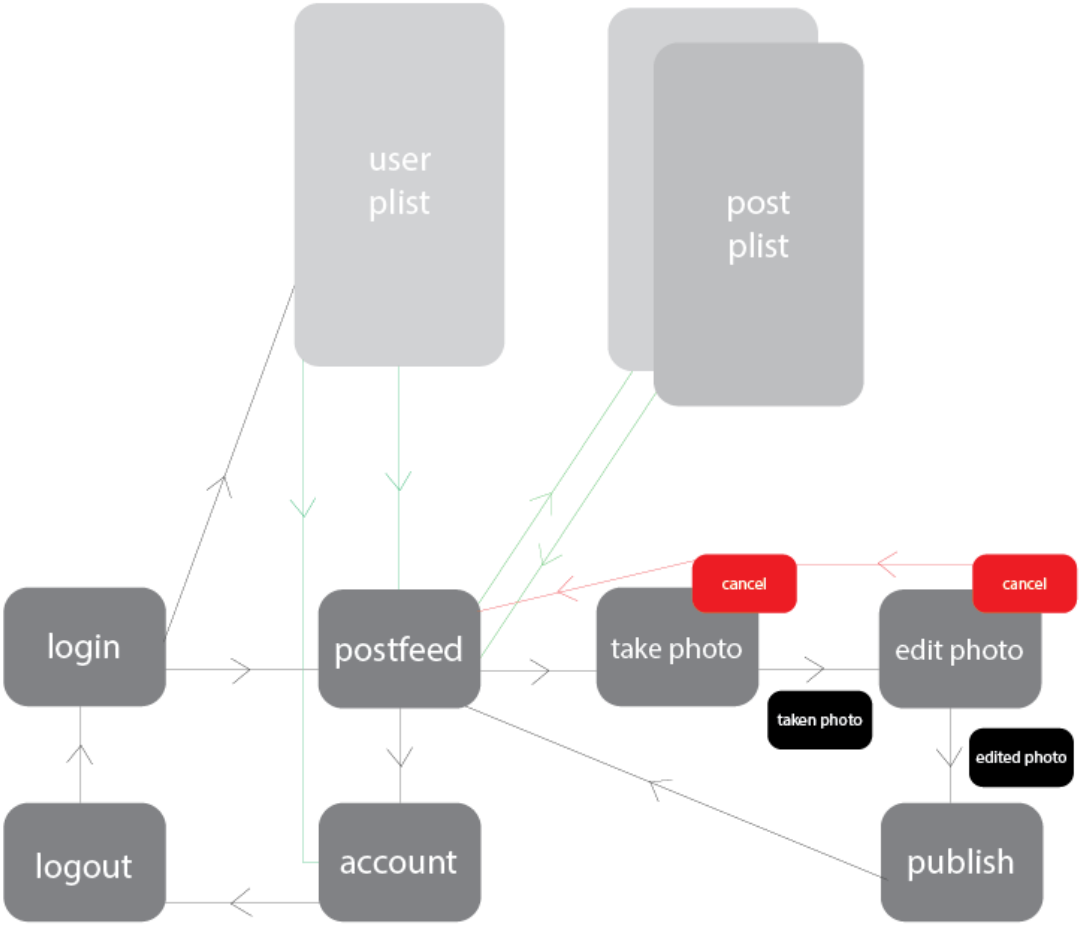
```
@interface post : NSObject

@property (strong, nonatomic) NSString *_id;
@property (strong, nonatomic) NSString *campaignIdentifier;
@property (strong, nonatomic) NSString *title;
@property (strong, nonatomic) NSString *description;
@property (strong, nonatomic) NSString *photoURL;
@property (strong, nonatomic) NSString *created;
@property (strong, nonatomic) NSString *authorFirstName;
@property (strong, nonatomic) NSString *authorLastName;

//user rating should be extended here

- (id)initWithId:(NSString *)_id andCampaignIdentifier:(NSString *)campaignIdentifier andTitle:(NSString *)title
    andDescription:(NSString *)description andPhotoURL:(NSString *)photoURL andCreated:(NSString *)created
    andAuthorFirstName:(NSString*)authorFirstName andAuthorLastName:(NSString*)authorLastName;

- (void) saveToPlist:(NSString *)plistURL;
+ (void) removeAllPosts:(NSString *)plistURL;
+ (NSMutableArray *)readFromPlist:(NSString *)plistURL;
+ (NSString *) getPlistURL;
+ (UIImage *)imageWithImage:(UIImage *)image scaledToSize:(CGSize)newSize;

@end
```

**user** class interface:

```
@interface user : NSObject

@property (strong, nonatomic) NSString *_id;
@property (strong, nonatomic) NSString *email;
@property (strong, nonatomic) NSString *firstName;
@property (strong, nonatomic) NSString *lastName;
@property (strong, nonatomic) NSString *bio;
@property (strong, nonatomic) NSString *level;
@property (strong, nonatomic) NSString *photoURL;
@property (strong, nonatomic) NSString *provider;
@property (strong, nonatomic) NSString *updated;
@property (strong, nonatomic) NSString *created;
@property (strong, nonatomic) NSString *campaign;
@property (strong, nonatomic) NSString *loginToken;

- (id)initWithId:(NSString *)_id andEmail:(NSString *)email andFirstName:(NSString *)firstName andLastName:(NSString *)
    lastName andBio:(NSString *)bio andLevel:(NSString *)level andPhotoURL:(NSString *)photoURL andProvider:(NSString *)
    provider andUpdated:(NSString *)updated andCreated:(NSString *)created andCampaign:(NSString *)campaign
    andLoginToken:(NSString *)loginToken;
- (void) saveToPlist:(NSString *)plistURL;
+ (void) removeFromPlist:(NSString *)plistURL;
+ (NSMutableArray *)readFromPlist:(NSString *)plistURL;
+ (NSString *) getPlistURL;

@end
```

The user class holds the token information (loginToken) that gives the logged-in user permission to send post requests to the web server.

# APPLICATION FLOW




Above the application flow is presented. It can be seen that from the login view a segue to the post feed is made. From there, logout can be achieved by segueing to the account view. Posting can be achieved by touching the camera icon, segueing to the camera view and then following two additional steps: passing the image data to the edit photo view and finally passing the edit image data to the publish view where it will be uploaded to the web server.

# TABLE VIEW IMPLEMENTATION

The homeViewController and implicitly the post feed make for the core part of the application. It serves as the 'home' of the application, from which all the other actions can be initiated.

## CELL INSTANTIATION

The homeViewController is the UITableViewController delegate as well as the UITableDataSource, which requires two methods to be created: **tableView:numberOfRowsInSection** and **tableView:cellForRowAtIndexPath:indexPath**.

The first returns the number or rows in each section. Having just one section, this method will count the number of posts that are loaded in the **loadedPostObjects** array.

```
#pragma mark - Table View methods
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section{
    return [loadedPostObjects count];
}
```

The other method is the core of the post feed, where all post data is added to the layout and the post image is loaded.

```
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)
    indexPath{
    //setup cell identifier and init cell
    static NSString *CellIdentifier = @"defaultPostCell";
    postTableViewCell *oneCell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];

    //load the corresponding post data
    post *currentPost = [loadedPostObjects objectAtIndex:indexPath.row];

    //cancel the cell selected background
    oneCell.selectionStyle = UITableViewCellSelectionStyleNone;
```

The first step is to setup cell reuse (by defining a cell identifier). This will allow the Table View to reuse cells that are not viewed at a certain moment. Afterwards, the corresponding post data is loaded from the **loadedPostObjects** array is loaded for convenience.

The cell selection is also disabled.

```
UIImage *placeholderPhoto = [UIImage imageNamed:@"image-placeholder"];
[oneCell.postImage setImage:placeholderPhoto];

//run async block
dispatch_queue_t queue = dispatch_queue_create("com.photoCampaign.postImageQueue", NULL);
dispatch_async(queue, ^{
    //load the image from photocampaign.net
    NSString *urlString = [[NSString alloc] initWithFormat:@"%@%@%@", REQUEST_URL, IMAGE_BASE_SLUG, [currentPost photoURL]];
    NSURL *photoURL = [[NSURL alloc] initWithString:urlString];
    UIImage *loadedPostPhoto = [UIImage imageWithData:[NSData dataWithContentsOfURL:photoURL]];

    //update UI after image load
    dispatch_async(dispatch_get_main_queue(), ^{
        //update the UI on the main thread
        [oneCell.postImage setClipsToBounds:YES];
        [oneCell.postImage setImage:loadedPostPhoto];
        //set imageview to scale image to it's frame
        [oneCell.postImage setContentMode:UIViewContentModeScaleAspectFill];

        //trigger display update
        [oneCell.postImage setNeedsDisplay];
        [oneCell.postImage setNeedsLayout];
    });
});
```

The next step is adding a 'blank' image from the local assets as a placeholder for the post image. After this task is completed, a new queue is created to handle the download of each of the cell photos. In the newly defined queue the image is downloaded and afterwards another asynchronous task is created in the main queue. This is done so because UI elements should be updated only in the main queue. After setting the downloaded image as the post image and setting a few additional parameters, the cell is updated.

```
//update post author label
oneCell.postAuthor.text = [NSString stringWithFormat:@"%@ %@", [currentPost authorFirstName], [currentPost authorLastName]];

//load relative time of post
    //convert the JSON date first
    NSDateFormatter *dateFormatter = [[NSDateFormatter alloc] init];
    [dateFormatter setDateFormat:@"yyyy-MM-dd'T'HH:mm:ss.SSSZ"];
    NSDate *postCreatedDate = [dateFormatter dateFromString:[currentPost created]];
    //convert to relative post date
    NSString *relativePostDate = postCreatedDate.timeAgoSinceNow;
//update label
oneCell.postDate.text = relativePostDate;

//update title of the post
oneCell.postTitle.text = [currentPost title];

return oneCell;
```

Finally, the author information is updated as well as the relative date and the post title. The final cell is then returned.

## SORTING CELLS

The post data in the cell is sorted beforehand to make sure it's arranged by date (most recent first).

The sorting is done in the method **reloadPostData** and it re-orders the **loadedPostObjects** as a result.

```
//sort elements by 'created' date
NSSortDescriptor *sortDescriptor;
sortDescriptor = [[NSSortDescriptor alloc] initWithKey:@"created"
                                             ascending:NO];
NSMutableArray *sortDescriptors = [NSMutableArray arrayWithObject:sortDescriptor];
NSArray *sortedArray;
sortedArray = [loadedPostObjects sortedArrayUsingDescriptors:sortDescriptors];
NSMutableArray *finalPostData = [[NSMutableArray alloc] initWithArray:sortedArray];
loadedPostObjects = finalPostData;
```

## PULL TO REFRESH

Pull to refresh is implemented in the viewDidLoad method by creating a new UIRefreshControl with the target self and **httpLoadPostData** as a action.

```
//setup refresh
UIRefreshControl *refresh = [[UIRefreshControl alloc] init];
[refresh addTarget:self action:@selector(httpLoadPostData) forControlEvents:UIControlEventValueChanged];
self.refreshControl = refresh;
```

When performing a 'pull to refresh', the designated method send a get request to the REST API and downloads all the new JSON data.

```
[manager GET:loginString parameters:params
     success:^(AFHTTPRequestOperation *operation, id responseObject) {
         //save post data and then trigger child table view population method
         for(NSArray* onePost in responseObject){
             //instantiate post object
             post* postToSave = [[post alloc] initWithId:[onePost valueForKey:@"_id"] andCampaignIdentifier:[[onePost
                 valueForKey:@"campaignObject" ] valueForKey:@"identifier"] andTitle:[onePost valueForKey:@"title"]
                 andDescription:[onePost valueForKey:@"description"]  andPhotoURL:[onePost valueForKey:@"photoURL"] andCreated:
                 [onePost valueForKey:@"created"] andAuthorFirstName:[[onePost valueForKey:@"owner"] valueForKey:@"firstName"]
                 andAuthorLastName:[[onePost valueForKey:@"owner"] valueForKey:@"lastName"]];

             //get post url
             NSString* postPlistURL = [post getPlistURL];
             [postToSave saveToPlist:postPlistURL];
         }

         [self reloadPostFeed];

         //log JSON response
         //NSLog(@"Campaign success JSON: %@", responseObject);
     }
```

Post Objects are instantiated and saved for each of the JSON objects. The post class checks if a post is already saved locally by comparing the existing '_id" and does not overwrite existing data:

```
if(![itemsHolder valueForKey:self._id]){
    //add to dictionary only if not already present
    //adding self as in the current instance of the post
    [itemsHolder setObject:self forKey:[NSString stringWithFormat:@"%@%@", postTimestamp, self._id]];
}
```

If the post feed cannot be downloaded from the web server, a error alert will show up, but all the posts downloaded until that moment will be shown.

```
failure:^(AFHTTPRequestOperation *operation, NSError *error) {
    UIAlertView *accountAlert = [[UIAlertView alloc] initWithTitle:@"Problem reading feed" message:@"Cannot read the campaign post feed.
        Please make sure there is an active internet connection" delegate:self cancelButtonTitle:@"Okay" otherButtonTitles:nil, nil];

    [accountAlert show];

    //NSLog(@"Campaign feed Error: %@", error);
}
```

# PHOTO EDITING IMPLEMENTATION

Photo editing is a resource intensive operation that can cause frequent application crashes due to high memory usage. When applying effects on 8MP photos memory warnings are unavoidable. However, this application resizes the photos before applying any effects (the size is not needed for web or iOS).

In the **photoeditViewController**, the viewDidLoad method contains instructions to resize the taken photo to half:

```
    //apply photo (resized to half)
    self.photo = self.photoInfo[UIImagePickerControllerOriginalImage];
    UIImage *resizedPhoto = [post imageWithImage:self.photo scaledToSize:CGSizeMake(self.photo.size.width/2, self.photo.size.height/2)];
    self.photo = resizedPhoto;
```

Afterwards, instead of applying effects on button press, each effect-image is generated once at initialisation. The buttons for switching effects are in fact just changing the photo in the layout's UIImageView.

To save the photos according to their effects, a array with photos is instantiated and saved in the **allImages** property.

```
//define filtered images
UIImage *tonalImage = [UIImage new];
UIImage *noirImage = [UIImage new];
UIImage *fadeImage = [UIImage new];
UIImage *processImage = [UIImage new];
UIImage *monoImage = [UIImage new];
UIImage *transferImage = [UIImage new];
UIImage *sepiaImage = [UIImage new];
self.allImages = [[NSMutableArray alloc] initWithArray: @[tonalImage, noirImage, fadeImage, processImage, monoImage, transferImage, sepiaImage]];
```

The same will be done with the photo effects, and then they will be looped through. Each loop will be made in it's own async queue, as follows:

```
- (void)loadPhotosWithEffects{
    CIFilter *sepiaFilter = [CIFilter filterWithName:@"CISepiaTone"];
    CIFilter *transferFilter = [CIFilter filterWithName:@"CIPhotoEffectTransfer"];
    CIFilter *noirFilter = [CIFilter filterWithName:@"CIPhotoEffectNoir"];
    CIFilter *tonalFilter = [CIFilter filterWithName:@"CIPhotoEffectTonal"];
    CIFilter *fadeFilter = [CIFilter filterWithName:@"CIPhotoEffectFade"];
    CIFilter *processFilter = [CIFilter filterWithName:@"CIPhotoEffectProcess"];
    CIFilter *monoFilter = [CIFilter filterWithName:@"CIPhotoEffectMono"];

    NSArray *allFilters = [[NSArray alloc] initWithObjects: tonalFilter, noirFilter, fadeFilter, processFilter, monoFilter, transferFilter, sepiaFilter,
        nil];

    //run async image load
    dispatch_queue_t queue = dispatch_queue_create("com.photoCampaign.postImageQueue", NULL);
    int filterPosition = 0;
    for(CIFilter *filter in allFilters){
        dispatch_async(queue, ^{
            [self.allImages setObject:[self setFilter:filter] atIndexedSubscript:filterPosition];
        });

        filterPosition ++;
    }
}
```

The queue calls the **setFilter** method and passed the current filter. The **setFilter** creates a filtered image which is written in the array at the corresponding position.

```
@interface photoeditViewController (){
    CIContext *currentContext;
    CIImage *filterResult;
    UIImage *filteredUIImage;
}
```

```
- (UIImage *)setFilter:(CIFilter*)currentFilter{
    currentContext = [CIContext contextWithOptions:nil];
    [currentFilter setValue:[[CIImage alloc] initWithImage:self.photo] forKey:kCIInputImageKey];
    //[filter setValue:@0.8f forKey:kCIInputIntensityKey];
    filterResult = [currentFilter valueForKey:kCIOutputImageKey];
    CGRect extent = [filterResult extent];
    CGImageRef cgImageRef = [currentContext createCGImage:filterResult fromRect:extent];
    filteredUIImage = [UIImage imageWithCGImage:cgImageRef scale:1 orientation:self.photo.imageOrientation];
    return filteredUIImage;
}
```

The **setFilter** method overwrites the CIContext, CIImage and UIImage for each filter operation. This makes it memory efficient and faster to apply all the filters at initialisation. The filtered image uses the rotation of the original photo and the scale of 1.

# KEYBOARD HIDING AND RETURN KEY

The keyboard hiding implementation is present in the views that have UITextFields. The login screen includes a custom implementation of the return key which (if on the email field) goes to the next UITextField and (if on the password field) submits the login information and hides the keyboard.

```
#pragma mark - Keyboard methods
- (BOOL)textFieldShouldReturn:(UITextField *)textField{
    if(textField == self.emailInput){
        [self.passwordInput becomeFirstResponder];
    }
    else{
        [textField resignFirstResponder];
        //call sign in IBAction
        [self signInAction:nil];
    }

    return YES;
}
```

Being a **UITextFieldDelegate**, the **loginViewController** can implement the **textFieldShouldReturn:** method. This method handles the custom return key action described above.

The second method that handles the keyboard hiding is **touchesBegan:withEvent**. In this method checks are done to see if either of the text fields are currently a **firstResponder**. If this is true, the keyboard will be hidden by calling **resignFirstResponder** on the currently active UITextField.

# PODS AND THIRD PARTY LIBRARIES

This project uses **cocoaPods** for installing third party libraries. This extension is similar to a package manager (npm - node package manager) and allows adding library dependencies to a project with ease.

Two libraries are used for this application:

**AFNetworking** - this library extends the NSURLConnection + NSOperation provided by Apple and makes sending web request (and reading their JSON response) a pleasant task. The library is used for sending all requests in this app.

**DateTools** - this library extends the date formatting algorithms and allows easily formatting dates created by JavaScript (ISO 8601) into relative times. (Among many more other extensions)

# CONCLUSION

Concluding this report, the demo video is available with a more detailed view of the platform:

youtube.com/watch?v=xlu2rq1LZaY