```
+
+ - (void) leftButtonPressedOnToolbar:(BLCCameraToolbar *)toolbar {
+     AVCaptureDeviceInput *currentCameraInput = self.session.inputs.firstObject;
+
+     NSArray *devices = [AVCaptureDevice devicesWithMediaType:AVMediaTypeVideo];
+
+     if (devices.count > 1) {
+         NSUInteger currentIndex = [devices indexOfObject:currentCameraInput.device];
+         NSUInteger newIndex = 0;
+
+         if (currentIndex < devices.count - 1) {
+             newIndex = currentIndex + 1;
+         }
+
+         AVCaptureDevice *newCamera = devices[newIndex];
+         AVCaptureDeviceInput *newVideoInput = [[AVCaptureDeviceInput alloc] initWithDevice:newCamera error:nil];
+
+         if (newVideoInput) {
+             UIView *fakeView = [self.imagePreview snapshotViewAfterScreenUpdates:YES];
+             fakeView.frame = self.imagePreview.frame;
+             [self.view insertSubview:fakeView aboveSubview:self.imagePreview];
+
+             [self.session beginConfiguration];
+             [self.session removeInput:currentCameraInput];
+             [self.session addInput:newVideoInput];
+             [self.session commitConfiguration];
+
+             [UIView animateWithDuration:0.2 delay:0 options:UIViewAnimationOptionCurveEaseInOut animations:^{
+                 fakeView.alpha = 0;
+             } completion:^(BOOL finished) {
+                 [fakeView removeFromSuperview];
+             }];
+         }
+     }
+ }
```

We get the current input as well as an array of all possible video devices. This is typically 2 (front camera and rear camera).

If there's more than one possible device, we switch to the next one (or to the beginning if we're at the end).

Once we get our new device, we'll try to create an input for it. Again, it's possibly but unlikely that this will fail.

If creating the input succeeds, we make a nice dissolve effect.

Responding to the Right Camera Toolbar Button

The right camera toolbar button will open a different view to allow the user to select a photo from their library.

We'll leave it unimplemented for this checkpoint:

BLCCameraViewController.m

```
+ - (void) rightButtonPressedOnToolbar:(BLCCameraToolbar *)toolbar {
+     NSLog(@"Photo library button pressed.");
+ }
+
```

Prerequisites to Taking Pictures

Before we can take a photo, we'll need to write a few methods for rotating, resizing, and cropping images.

Because we will use these methods elsewhere, we'll take this opportunity to show you a fun feature of Objective-C called **categories.**

Introducing Categories

From **Programming with Objective-C: Customizing Existing Classes**:

> *Sometimes, you may find that you wish to extend an existing class by adding behavior that is useful only in certain situations. [...] In situations like this, it doesn't always make sense to add the utility behavior to the original, primary class interface. [...] Instead, Objective-C allows you to add your own methods to existing classes through categories.*
>
> ***Deep Dive: Introducing Categories***

Now let's get back to our camera app, and add some additional functionality to **`UIImage`**.

Adding a Category on **`UIImage`**

Here's how to create your first category:

1. Press **File** > **New** > **File...**.
2. In **iOS** > **Cocoa Touch**, select **Objective-C Category** and press **Next**.
3. Call your category **`BLCImageUtilities`** and make it a category on **`UIImage`**.
4. Press **Create**.

Open up **`UIImage+BLCImageUtilities.h`**. We're going to create three methods:

UIImage+BLCImageUtilities.h

```
+ @interface UIImage (BLCImageUtilities)
+
+ - (UIImage *) imageWithFixedOrientation;
+ - (UIImage *) imageResizedToMatchAspectRatioOfSize:(CGSize)size;
+ - (UIImage *) imageCroppedToRect:(CGRect)cropRect;
+
+ @end
```

Now let's switch to the **`.m`** file for implementation.

Fixing Orientation

`UIImage`s are often stored rotated and/or mirrored.

This method will inspect the image's **`imageOrientation`** property, which lets us know if the image has been rotated or mirrored. We'll then us this information to flip or rotate the image as necessary.

Here's how the code looks:

UIImage+BLCImageUtilities.m

```
@implementation UIImage (BLCImageUtilities)

+ - (UIImage *) imageWithFixedOrientation {
+     // Do nothing if the orientation is already correct
+     if (self.imageOrientation == UIImageOrientationUp) return [self copy];
+
+     // We need to calculate the proper transformation to make the image upright.
+     // We do it in 2 steps: Rotate if Left/Right/Down, and then flip if Mirrored.
+     CGAffineTransform transform = CGAffineTransformIdentity;
+
+     switch (self.imageOrientation) {
+         case UIImageOrientationDown:
+         case UIImageOrientationDownMirrored:
+             transform = CGAffineTransformTranslate(transform, self.size.width, self.size.height);
```

```
+
+        case UIImageOrientationLeft:
+        case UIImageOrientationLeftMirrored:
+            transform = CGAffineTransformTranslate(transform, self.size.width, 0);
+            transform = CGAffineTransformRotate(transform, M_PI_2);
+            break;
+
+        case UIImageOrientationRight:
+        case UIImageOrientationRightMirrored:
+            transform = CGAffineTransformTranslate(transform, 0, self.size.height);
+            transform = CGAffineTransformRotate(transform, -M_PI_2);
+            break;
+        case UIImageOrientationUp:
+        case UIImageOrientationUpMirrored:
+            break;
+    }
+
+    switch (self.imageOrientation) {
+        case UIImageOrientationUpMirrored:
+        case UIImageOrientationDownMirrored:
+            transform = CGAffineTransformTranslate(transform, self.size.width, 0);
+            transform = CGAffineTransformScale(transform, -1, 1);
+            break;
+
+        case UIImageOrientationLeftMirrored:
+        case UIImageOrientationRightMirrored:
+            transform = CGAffineTransformTranslate(transform, self.size.height, 0);
+            transform = CGAffineTransformScale(transform, -1, 1);
+            break;
+        case UIImageOrientationUp:
+        case UIImageOrientationDown:
+        case UIImageOrientationLeft:
+        case UIImageOrientationRight:
+            break;
+    }
+
+    // Now we draw the underlying CGImage into a new context, applying the transform
+    // calculated above.
+    CGFloat scaleFactor = self.scale;
+
+
+    CGContextRef ctx = CGBitmapContextCreate(NULL,
+                                             self.size.width * scaleFactor,
+                                             self.size.height * scaleFactor,
+                                             CGImageGetBitsPerComponent(self.CGImage),
+                                             0,
+                                             CGImageGetColorSpace(self.CGImage),
+                                             CGImageGetBitmapInfo(self.CGImage));
+
+    CGContextScaleCTM(ctx, scaleFactor, scaleFactor);
+
+    CGContextConcatCTM(ctx, transform);
+    switch (self.imageOrientation) {
+        case UIImageOrientationLeft:
+        case UIImageOrientationLeftMirrored:
+        case UIImageOrientationRight:
+        case UIImageOrientationRightMirrored:
+            CGContextDrawImage(ctx, CGRectMake(0,0, self.size.height, self.size.width), self.CGImage);
+            break;
+
+        default:
+            CGContextDrawImage(ctx, CGRectMake(0,0, self.size.width, self.size.height), self.CGImage);
+            break;
+    }
+
+    // Create a new UIImage from the drawing context
+    CGImageRef cgimg = CGBitmapContextCreateImage(ctx);
+    UIImage *img = [UIImage imageWithCGImage:cgimg scale:scaleFactor orientation:UIImageOrientationUp];
+    CGContextRelease(ctx);
+    CGImageRelease(cgimg);
+    return img;
```

> *This code, and the similar code in the other two* **`UIImage`** *categories, are written in C, not Objective-C, so don't worry that they look a little different. You won't need to delve into C much as an iOS developer, so we won't spend a ton of time on it.*

`transform` holds an "**affine transformation matrix**". In layman's terms, it's a grid of numbers (like a spreadsheet) that describes how to rotate, flip, and scale a 2D image. The transform functions update the affine transform (**`transform`**) to reflect how you want to rotate/flip/scal the image.

Once we've created **`transform`**, we:

1. Create a **`CGContextRef`**(a "bitmap graphics context"), which is like a blank sheet of paper you can draw on
2. Scale the image to support Retina displays
3. Apply **`transform`** to the drawing context with **`CGContextConcatCTM`**, and
4. Draw the transformed image from the graphics context using **`CGContextDrawImage`**.

We then create the **`UIImage`** to return from the **`CGImageRef`**.

Since Automatic Reference Counting (ARC) doesn't support C, we release our objects when we're done with them using **`CGContextRelease`** and **`CGImageRelease`**.

Finally, we return the **`UIImage`**.

> *This probably feels overwhelming, but keep reading! You'll see that the same functions are typically used in the same order, just in a slightly different way. This will help you understand what these functions do.*

Resizing images to match the aspect ratio

The aspect ratio of the iOS device's screen is not the same as the aspect ratio of the iOS device's camera.

We'll need to resize an image to the aspect ratio of the screen in order to make the cropping rectangle accurate.

Here's the code:

UIImage+BLCImageUtilities.m

```
+     CGFloat horizontalRatio = size.width / self.size.width;
+     CGFloat verticalRatio = size.height / self.size.height;
+     CGFloat ratio = MAX(horizontalRatio, verticalRatio);
+     CGSize newSize = CGSizeMake(self.size.width * ratio * self.scale, self.size.height * ratio * self.scale);
+
+     CGRect newRect = CGRectIntegral(CGRectMake(0, 0, newSize.width, newSize.height));
+     CGImageRef imageRef = self.CGImage;
+
+     CGContextRef ctx = CGBitmapContextCreate(NULL,
+                                              newRect.size.width,
+                                              newRect.size.height,
+                                              CGImageGetBitsPerComponent(self.CGImage),
+                                              0,
+                                              CGImageGetColorSpace(self.CGImage),
+                                              CGImageGetBitmapInfo(self.CGImage));
+
+     // Draw into the context; this scales the image
+     CGContextDrawImage(ctx, newRect, imageRef);
+
+     // Get the resized image from the context and a UIImage
+     CGImageRef newImageRef = CGBitmapContextCreateImage(ctx);
+     UIImage *newImage = [UIImage imageWithCGImage:newImageRef scale:self.scale orientation:UIImageOrientationUp];
+
+     // Clean up
+     CGContextRelease(ctx);
+     CGImageRelease(newImageRef);
+
+     return newImage;
+ }
```

We calculate the aspect ratio (**`ratio`**) and use that to calculate the size of the resized image (**`newSize`**). We use that size to create a new bitmap drawing context in the appropriate size, and then draw the image into it. Finally, we use the same steps as before to get a **`CGImageRef`** and convert that to a **`UIImage`**. (We also perform the same memory cleanup).

Cropping

Finally, one more action we'll need to perform: cropping.

UIImage+BLCImageUtilities.m

```
+ - (UIImage *) imageCroppedToRect:(CGRect)cropRect {
+     cropRect.size.width *= self.scale;
+     cropRect.size.height *= self.scale;
+     cropRect.origin.x *= self.scale;
+     cropRect.origin.y *= self.scale;
+
+     CGImageRef imageRef = CGImageCreateWithImageInRect(self.CGImage, cropRect);
+     UIImage *image = [UIImage imageWithCGImage:imageRef scale:self.scale orientation:self.imageOrientation];
+     CGImageRelease(imageRef);
+     return image;
+ }

 @end
```

This implementation is the simplest of the three because **`CGImageCreateWithImageInRect`** does nearly all of the work for us. We simply use that to create the cropped image, convert to **`UIImage`**, release the memory, and return the **`UIImage`**.

Putting it All Together: Taking Pictures

Let's go back to **`BLCCameraViewController.m`** so we can take our picture.

First, import our awesome new category at the top of the file:

```
+ #import "UIImage+BLCImageUtilities.h"
```

Here's the code for handling the camera button press. We'll walk through it on the other side.

```
+ - (void) cameraButtonPressedOnToolbar:(BLCCameraToolbar *)toolbar {
+     AVCaptureConnection *videoConnection;
+
+     // Find the right connection object
+     for (AVCaptureConnection *connection in self.stillImageOutput.connections) {
+         for (AVCaptureInputPort *port in connection.inputPorts) {
+             if ([port.mediaType isEqual:AVMediaTypeVideo]) {
+                 videoConnection = connection;
+                 break;
+             }
+         }
+         if (videoConnection) { break; }
+     }
+
+     [self.stillImageOutput captureStillImageAsynchronouslyFromConnection:videoConnection completionHandler: ^(CMSampleBufferRef ima
+         if (imageSampleBuffer) {
+             NSData *imageData = [AVCaptureStillImageOutput jpegStillImageNSDataRepresentation:imageSampleBuffer];
+             UIImage *image = [UIImage imageWithData:imageData scale:[UIScreen mainScreen].scale];
+             image = [image imageWithFixedOrientation];
+             image = [image imageResizedToMatchAspectRatioOfSize:self.captureVideoPreviewLayer.bounds.size];
+
+             UIView *leftLine = self.verticalLines.firstObject;
+             UIView *rightLine = self.verticalLines.lastObject;
+             UIView *topLine = self.horizontalLines.firstObject;
+             UIView *bottomLine = self.horizontalLines.lastObject;
+
+             CGRect gridRect = CGRectMake(CGRectGetMinX(leftLine.frame),
+                                          CGRectGetMinY(topLine.frame),
+                                          CGRectGetMaxX(rightLine.frame) - CGRectGetMinX(leftLine.frame),
+                                          CGRectGetMinY(bottomLine.frame) - CGRectGetMinY(topLine.frame));
+
+             CGRect cropRect = gridRect;
+             cropRect.origin.x = (CGRectGetMinX(gridRect) + (image.size.width - CGRectGetWidth(gridRect)) / 2);
+
+             image = [image imageCroppedToRect:cropRect];
+
+             dispatch_async(dispatch_get_main_queue(), ^{
+                 [self.delegate cameraViewController:self didCompleteWithImage:image];
+             });
+         } else {
+             dispatch_async(dispatch_get_main_queue(), ^{
+                 UIAlertView *alert = [[UIAlertView alloc] initWithTitle:error.localizedDescription message:error.localizedRecoveryS
+                 [alert show];
+             });
+
+         }
+     }];
+ }
```

First, we need to find the correct **`AVCaptureConnection`**, which represents the *input - session - output* connection. We pass this connection to the output object (so it knows which input to retrieve from), and it returns a completion block with the image in the form of a **`CMSampleBufferRef`**, which can contain a variety of media-related data.

Fortunately, we know it's a JPEG still image, so we can easily convert it to an **`NSData`** object and then to a **`UIImage`** object.

After that, we fix the orientation, and scale it using 2/3 of the **`UIImage`** category methods we just wrote.

Then, we calculate the **`CGRect`** of the white square (**`gridRect`**), and make sure it's centered (**`cropRect`**). We pass that to the final method to crop the image in the shape of a square.

Finally, we call the delegate method with the image. The camera button should now capture the correct image.

Adding a Camera Button to the Images View Controller

Let's venture back into **BLCImagesTableViewController.m**, where we'll add a camera button.

Import the new view controller we just finished, and declare that we'll conform to its delegate:

BLCImagesTableViewController.m

```
#import "BLCMediaTableViewCell.h"
#import "BLCMediaFullScreenViewController.h"
#import "BLCMediaFullScreenAnimator.h"
+ #import "BLCCameraViewController.h"

- @interface BLCImagesTableViewController () <BLCMediaTableViewCellDelegate, UIViewControllerTransitioningDelegate>
+ @interface BLCImagesTableViewController () <BLCMediaTableViewCellDelegate, UIViewControllerTransitioningDelegate, BLCCameraViewCont

@property (nonatomic, weak) UIImageView *lastTappedImageView;
@property (nonatomic, weak) UIView *lastSelectedCommentView;
```

In **viewDidLoad**, check if any photo capabilities at all are available, and if so, add a camera button:

BLCImagesTableViewController.m

```
    self.tableView.keyboardDismissMode = UIScrollViewKeyboardDismissModeInteractive;

+     if ([UIImagePickerController isSourceTypeAvailable:UIImagePickerControllerSourceTypeCamera] ||
+         [UIImagePickerController isSourceTypeAvailable:UIImagePickerControllerSourceTypeSavedPhotosAlbum]) {
+         UIBarButtonItem *cameraButton = [[UIBarButtonItem alloc] initWithBarButtonSystemItem:UIBarButtonSystemItemCamera target:sel
+         self.navigationItem.rightBarButtonItem = cameraButton;
+     }
+
    [[NSNotificationCenter defaultCenter] addObserver:self
                                             selector:@selector(keyboardWillShow:)
                                                 name:UIKeyboardWillShowNotification
```

Present the view controller within a navigation controller when the button is pressed, and dismiss it with a note when the delegate method is called:

BLCImagesTableViewController.m

```
+ #pragma mark - Camera and BLCCameraViewControllerDelegate
+
+ - (void) cameraPressed:(UIBarButtonItem *) sender {
+     BLCCameraViewController *cameraVC = [[BLCCameraViewController alloc] init];
+     cameraVC.delegate = self;
+     UINavigationController *nav = [[UINavigationController alloc] initWithRootViewController:cameraVC];
+     [self presentViewController:nav animated:YES completion:nil];
+     return;
+ }
+
+ - (void) cameraViewController:(BLCCameraViewController *)cameraViewController didCompleteWithImage:(UIImage *)image {
+     [cameraViewController dismissViewControllerAnimated:YES completion:^{
+         if (image) {
+             NSLog(@"Got an image!");
+         } else {
+             NSLog(@"Closed without an image.");
+         }
+     }];
+ }
+
#pragma mark - Table view data source

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section
```

If you have an Apple Developer account, run the app on your iPhone or iPod Touch. You should be able to take a photo, like in the screenshot

Summary

You've learned quite a bit!

- How **AVFoundation** capture devices, inputs, and outputs work together to allow your app to capture multimedia
- How Categories can extend the functionality of a class, even if it's not yours
- How to use Core Graphics bitmap contexts to rotate, flip, crop, and resize images

Create a new **UIImage** category method with this method signature:

```
- (UIImage *) imageByScalingToSize:(CGSize)size andCroppingWithRect:(CGRect)rect;
```

Have it complete all of the related **UIImage** work that's currently done in the camera view controller, so that none of the other category methods need to be called there.

When you're satisfied with your project and the assignment, commit, merge and push your code to GitHub:

Terminal

```
$ git add .
$ git commit -m 'Added Camera View'
$ git checkout master
$ git merge taking-pictures
$ git push
```

Submit this checkpoint's assignment with links for your repo and commit.

assignment completed

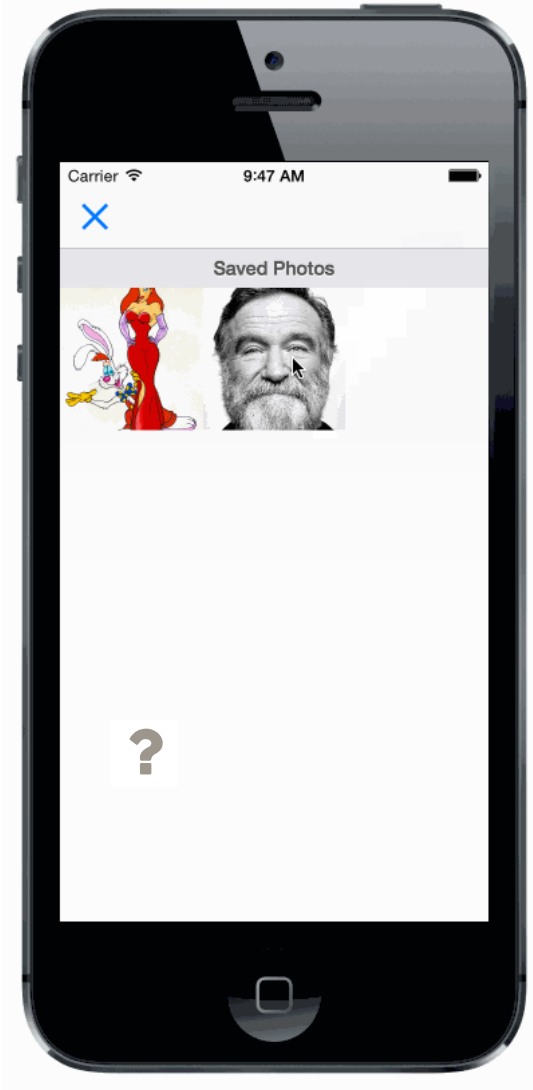
hello@bloc.io

Considering enrolling? (404) 480-2562

Partnership / Corporate Inquiries? (650) 741-5682

Roadmap · Previous Checkpoint
Jump to Submission · Next Checkpoir

Accessing the Image Library



Taking pictures with the camera was fun (and a little complicated), but sometimes we don't need to do that. Some users have their own images already stored on their device!

There's lots of ways that photos can get added to a device:

- By taking a picture with a camera (on iOS devices with cameras)
- By saving a picture from another source, like Safari
- By syncing a picture to the device from iTunes
- By subscribing to an iCloud Shared Photo Library

Giving our users access to these images will open up a whole new world of posting square-cropped photos of food and vacations for the worl to see.

In this checkpoint, you'll learn how to use **`ALAssetsLibrary`** to retrieve assets from the user's assets library (commonly referred to as their

We'll also modify and reuse some of our existing code to make it a lot less painful.

cd into your Blocstagram directory and make a new **git** branch:

Terminal

```
$ git checkout -b using-assets-library
```

Some Preliminary Work

We know the user will want to crop their image into a square. To avoid repeating lots of layout code in two places, let's start by moving our 3x grid of white-bordered squares into a new class.

Create a new **UIView** subclass called **BLCCropBox**.

The interface will be blank:

BLCCropBox.h

```
#import <UIKit/UIKit.h>

@interface BLCCropBox : UIView

@end
```

The **.m** file will have code ripped right out of **BLCCameraViewController** (the view controller we made in the prior checkpoint). Here's how it looks:

BLCCropBox.m

```
 #import "BLCCropBox.h"

+ @interface BLCCropBox ()
+
+ @property (nonatomic, strong) NSArray *horizontalLines;
+ @property (nonatomic, strong) NSArray *verticalLines;
+
+ @end
+
 @implementation BLCCropBox

 - (id)initWithFrame:(CGRect)frame
 {
     self = [super initWithFrame:frame];
     if (self) {
+         self.userInteractionEnabled = NO;
+
         // Initialization code
+         NSArray *lines = [self.horizontalLines arrayByAddingObjectsFromArray:self.verticalLines];
+         for (UIView *lineView in lines) {
+             [self addSubview:lineView];
+         }
     }
     return self;
 }

+ - (NSArray *) horizontalLines {
+     if (!_horizontalLines) {
+         _horizontalLines = [self newArrayOfFourWhiteViews];
+     }
+
+     return _horizontalLines;
+ }
```

```
+     if (!_verticalLines) {
+         _verticalLines = [self newArrayOfFourWhiteViews];
+     }
+
+     return _verticalLines;
+ }
+
+ - (NSArray *) newArrayOfFourWhiteViews {
+     NSMutableArray *array = [NSMutableArray array];
+
+     for (int i = 0; i < 4; i++) {
+         UIView *view = [UIView new];
+         view.backgroundColor = [UIColor whiteColor];
+         [array addObject:view];
+     }
+
+     return array;
+ }
+
+ - (void) layoutSubviews {
+     [super layoutSubviews];
+
+     CGFloat width = CGRectGetWidth(self.frame);
+     CGFloat thirdOfWidth = width / 3;
+
+     for (int i = 0; i < 4; i++) {
+         UIView *horizontalLine = self.horizontalLines[i];
+         UIView *verticalLine = self.verticalLines[i];
+
+         horizontalLine.frame = CGRectMake(0, (i * thirdOfWidth), width, 0.5);
+
+         CGRect verticalFrame = CGRectMake(i * thirdOfWidth, 0, 0.5, width);
+
+         if (i == 3) {
+             verticalFrame.origin.x -= 0.5;
+         }
+
+         verticalLine.frame = verticalFrame;
+     }
+ }
@end
```

There are only two minor differences between this, and the code we stole:

1. We set **`userInteractionEnabled`** to **`NO`** (since this view doesn't handle any touch events)
2. We replace the Y-coordinate in the layout code with **`0`**, since the view controller will position this box appropriately.

Let's update **`BLCCameraViewController.m`** to use this new view instead of all of those line.

Import the crop box class:

BLCCameraViewController.m

```
#import "UIImage+BLCImageUtilities.h"
+ #import "BLCCropBox.h"

@interface BLCCameraViewController () <BLCCameraToolbarDelegate, UIAlertViewDelegate>
```

Remove the old lines properties and add the crop box:

BLCCameraViewController.m

```
@property (nonatomic, strong) AVCaptureStillImageOutput *stillImageOutput;

- @property (nonatomic, strong) NSArray *horizontalLines;
- @property (nonatomic, strong) NSArray *verticalLines;
@property (nonatomic, strong) UIToolbar *topView;
@property (nonatomic, strong) UIToolbar *bottomView;
-
+ @property (nonatomic, strong) BLCCropBox *cropBox;
@property (nonatomic, strong) BLCCameraToolbar *cameraToolbar;
```

Create the crop box in **createViews**:

BLCCameraViewController.m

```
    self.topView = [UIToolbar new];
    self.bottomView = [UIToolbar new];
+     self.cropBox = [BLCCropBox new];
    self.cameraToolbar = [[BLCCameraToolbar alloc] initWithImageNames:@[@"rotate", @"road"]];
```

Add it to the view hierarchy (and remove the lines) in **addViewsToViewHierarchy**:

BLCCameraViewController.m

```
-     NSMutableArray *views = [@[self.imagePreview, self.topView, self.bottomView] mutableCopy];
+     NSMutableArray *views = [@[self.imagePreview, self.cropBox, self.topView, self.bottomView] mutableCopy];
-     [views addObjectsFromArray:self.horizontalLines];
-     [views addObjectsFromArray:self.verticalLines];
    [views addObject:self.cameraToolbar];
```

Delete the overridden getters (these are in **BLCCropBox** now):

BLCCameraViewController.m

```
- - (NSArray *) horizontalLines {
-     if (!_horizontalLines) {
-         _horizontalLines = [self newArrayOfFourWhiteViews];
-     }
-
-     return _horizontalLines;
- }
-
- - (NSArray *) verticalLines {
-     if (!_verticalLines) {
-         _verticalLines = [self newArrayOfFourWhiteViews];
-     }
-
-     return _verticalLines;
- }
-
- - (NSArray *) newArrayOfFourWhiteViews {
-     NSMutableArray *array = [NSMutableArray array];
-
-     for (int i = 0; i < 4; i++) {
-         UIView *view = [UIView new];
-         view.backgroundColor = [UIColor whiteColor];
-         [array addObject:view];
-     }
-
-     return array;
- }
```

Position the crop box, and remove the layout code for the old lines:

BLCCameraViewController.m

```
    self.bottomView.frame = CGRectMake(0, yOriginOfBottomView, width, heightOfBottomView);

-     CGFloat thirdOfWidth = width / 3;
-
-     for (int i = 0; i < 4; i++) {
-         UIView *horizontalLine = self.horizontalLines[i];
-         UIView *verticalLine = self.verticalLines[i];
-
-         horizontalLine.frame = CGRectMake(0, (i * thirdOfWidth) + CGRectGetMaxY(self.topView.frame), width, 0.5);
-
-         CGRect verticalFrame = CGRectMake(i * thirdOfWidth, CGRectGetMaxY(self.topView.frame), 0.5, width);
-
-         if (i == 3) {
-             verticalFrame.origin.x -= 0.5;
-         }
-
-         verticalLine.frame = verticalFrame;
-     }
+     self.cropBox.frame = CGRectMake(0, CGRectGetMaxY(self.topView.frame), width, width);
```

Our cropping logic no longer needs to calculate the grid's perimeter; instead it'll just use the crop box's frame:

BLCCameraViewController.m

```
            image = [image imageWithFixedOrientation];
            image = [image imageResizedToMatchAspectRatioOfSize:self.captureVideoPreviewLayer.bounds.size];

-             UIView *leftLine = self.verticalLines.firstObject;
-             UIView *rightLine = self.verticalLines.lastObject;
-             UIView *topLine = self.horizontalLines.firstObject;
-             UIView *bottomLine = self.horizontalLines.lastObject;
-
-             CGRect gridRect = CGRectMake(CGRectGetMinX(leftLine.frame),
-                                          CGRectGetMinY(topLine.frame),
-                                          CGRectGetMaxX(rightLine.frame) - CGRectGetMinX(leftLine.frame),
-                                          CGRectGetMinY(bottomLine.frame) - CGRectGetMinY(topLine.frame));
+             CGRect gridRect = self.cropBox.frame;

            CGRect cropRect = gridRect;
            cropRect.origin.x = (CGRectGetMinX(gridRect) + (image.size.width - CGRectGetWidth(gridRect)) / 2);
```

Creating a Controller that Can Crop

After a user selects an image, we'll want them to be able to pan around and zoom to crop the perfect square image.

Where have we already implemented panning and zooming images? That's right, in **`BLCMediaFullScreenViewController`**. Instead of reinventing the wheel, let's just subclass it and slightly modify its behavior.

Make two small changes to this class:

1. Move the **`media`** property to the **`.h`** file so that the subclass can access it.
2. Add a method to allow subclasses request the recalculation of the zoom scales.

The first is a simple cut and paste job:

Cut:

BLCMediaFullScreenViewController.m

```
@interface BLCMediaFullScreenViewController () <UIScrollViewDelegate>

- @property (nonatomic, strong) BLCMedia *media;
-
@property (nonatomic, strong) UITapGestureRecognizer *tap;
```

Paste:

BLCMediaFullScreenViewController.h

```
@property (nonatomic, strong) UIScrollView *scrollView;
@property (nonatomic, strong) UIImageView *imageView;

+ @property (nonatomic, strong) BLCMedia *media;
+
- (instancetype) initWithMedia:(BLCMedia *)media;
```

For the second, declare the method the in **`.h`** file:

BLCMediaFullScreenViewController.h

```
- (void) centerScrollView;

+ - (void) recalculateZoomScale;
+
@end
```

And implement it in the **`.m`** file:

BLCMediaFullScreenViewController.m

```
    [super viewWillLayoutSubviews];
    self.scrollView.frame = self.view.bounds;

+     [self recalculateZoomScale];
+ }
+
+ - (void) recalculateZoomScale {
    CGSize scrollViewFrameSize = self.scrollView.frame.size;
    CGSize scrollViewContentSize = self.scrollView.contentSize;

+     scrollViewContentSize.height /= self.scrollView.zoomScale;
+     scrollViewContentSize.width /= self.scrollView.zoomScale;
+
    CGFloat scaleWidth = scrollViewFrameSize.width / scrollViewContentSize.width;
    CGFloat scaleHeight = scrollViewFrameSize.height / scrollViewContentSize.height;
    CGFloat minScale = MIN(scaleWidth, scaleHeight);
```

> *The two lines we added which divide the size dimensions by* **`self.scrollView.zoomScale`** *allow subclasses to recalculate the zoom scale for scroll views that are zoomed out, which ours will be.*

Now, create a new **`BLCMediaFullScreenViewController`** subclass called **`BLCCropImageViewController`**.

Here's how the implementation will look, with a basic delegate pattern and an initializer:

BLCCropImageViewController.h

```
#import "BLCMediaFullScreenViewController.h"

+ @class BLCCropImageViewController;
+
+ @protocol BLCCropImageViewControllerDelegate <NSObject>
+
+ - (void) cropControllerFinishedWithImage:(UIImage *)croppedImage;
+
+ @end
+
+ @interface BLCCropImageViewController : BLCMediaFullScreenViewController
+
+ - (instancetype) initWithImage:(UIImage *)sourceImage;
+
+ @property (nonatomic, weak) NSObject <BLCCropImageViewControllerDelegate> *delegate;
+
+ @end
```

As you may have guessed from the interface, the way this view controller will work is straightforward:

1. Another controller will pass it a **UIImage** and set itself as the crop controller's delegate.
2. The user will size and crop the image, and the controller will pass a new, cropped **UIImage** back to its delegate.

In the **.m** file, import the crop box we made above, **BLCMedia**, and our **UIImage** utilities category. Also, add properties for the crop box and a **BOOL** called **hasLoadedOnce** (you'll see that in a bit):

BLCCropImageViewController.m

```
#import "BLCCropImageViewController.h"
+ #import "BLCCropBox.h"
+ #import "BLCMedia.h"
+ #import "UIImage+BLCImageUtilities.h"
+
@interface BLCCropImageViewController ()

+ @property (nonatomic, strong) BLCCropBox *cropBox;
+ @property (nonatomic, assign) BOOL hasLoadedOnce;
+
+ @end
```

Let's start with the initializer. Since our superclass (**BLCMediaFullScreenViewController**) uses a **BLCMedia** item, we'll create a new **BLCMedia** item from the image. We'll also initialize the crop box:

BLCCropImageViewController.m

```
@implementation BLCCropImageViewController

+ - (instancetype) initWithImage:(UIImage *)sourceImage {
+     self = [super init];
+
+     if (self) {
+         self.media = [[BLCMedia alloc] init];
+         self.media.image = sourceImage;
+
+         self.cropBox = [BLCCropBox new];
+     }
+
+     return self;
+ }
```

In **viewDidLoad**, we don't have much to do since the superclass takes care of most of the work. We need to:

- Set **clipsToBounds** to **YES** so the crop image doesn't overlap other controllers during navigation controller transitions
- Add the crop box to the view hierarchy
- Create a "crop image" button in the navigation bar

BLCCropImageViewController.m

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view.

+     self.view.clipsToBounds = YES;
+
+     [self.view addSubview:self.cropBox];
+
+     UIBarButtonItem *rightButton = [[UIBarButtonItem alloc] initWithTitle:NSLocalizedString(@"Crop", @"Crop command") style:UIBarBu
+
+     self.navigationItem.title = NSLocalizedString(@"Crop Image", nil);
+     self.navigationItem.rightBarButtonItem = rightButton;
+
+     self.automaticallyAdjustsScrollViewInsets = NO;
+
+     self.view.backgroundColor = [UIColor colorWithWhite:0.8 alpha:1];
}
```

Our implementation of **`viewWillLayoutSubviews`** is only responsible for laying out the views we've added, and modifying any superclass behavior. As with **`viewDidLoad`**, the superclass handles the rest.

Our implementation:

- sizes and centers **`cropBox`**.
- reduces the scroll view's frame to the same as the crop box's
- disables **`clipsToBound`** so the user can still see the image outside the crop box's

After changing the scroll view's frame, we must recalculate the zoom scale.

Additionally, on the first load, we zoom the picture all the way out (the superclass zooms all the way in).

Here's how it looks:

BLCCropImageViewController.m

```
+ - (void) viewWillLayoutSubviews {
+     [super viewWillLayoutSubviews];
+
+     CGRect cropRect = CGRectZero;
+
+     CGFloat edgeSize = MIN(CGRectGetWidth(self.view.frame), CGRectGetHeight(self.view.frame));
+     cropRect.size = CGSizeMake(edgeSize, edgeSize);
+
+     CGSize size = self.view.frame.size;
+
+     self.cropBox.frame = cropRect;
+     self.cropBox.center = CGPointMake(size.width / 2, size.height / 2);
+     self.scrollView.frame = self.cropBox.frame;
+     self.scrollView.clipsToBounds = NO;
+
+     [self recalculateZoomScale];
+
+     if (self.hasLoadedOnce == NO) {
+         self.scrollView.zoomScale = self.scrollView.minimumZoomScale;
+         self.hasLoadedOnce = YES;
+     }
+ }
```

All we need to do now is handle the cropping itself. This is pretty painless because of the **`imageWithFixedOrientation`** and **`imageCroppedToRect:`** category methods we created in the prior checkpoint. We just create the rect based on the scroll view's panned and zoomed location, and pass it into this method. Finally, we call the delegate:

```
+    CGRect visibleRect;
+    float scale = 1.0f / self.scrollView.zoomScale / self.media.image.scale;
+    visibleRect.origin.x = self.scrollView.contentOffset.x * scale;
+    visibleRect.origin.y = self.scrollView.contentOffset.y * scale;
+    visibleRect.size.width = self.scrollView.bounds.size.width * scale;
+    visibleRect.size.height = self.scrollView.bounds.size.height * scale;
+
+    UIImage *scrollViewCrop = [self.media.image imageWithFixedOrientation];
+    scrollViewCrop = [scrollViewCrop imageCroppedToRect:visibleRect];
+
+    [self.delegate cropControllerFinishedWithImage:scrollViewCrop];
+ }
+
+ @end
```

And we're done! Let's move on to displaying the image library, so the user can pick an image to crop.

Accessing and Displaying the Image Library

`ALAssetsLibrary` will let us get at the images, but how should we display them?

Although there are countless ways to display images, we'll use a class called **`UICollectionView`** to display them in a grid, similar to the UI in the Photos app.

About **`UICollectionView`**

`UICollectionView` mimics the patterns of **`UITableView`**, but in a more flexible way, which allows us to make any layout possible. While table views are constrained to a single-column vertical list, the display of information in a collection view can take any form that you can imagine.

Here are some more similarities between **`UITableView`** and **`UICollectionView`**:

- Collection views have a *delegate* and a *data source*; you respond to these methods to give the collection view information about cells to display.
- Collection views have a handy view controller subclass (**`UICollectionViewController`**) that makes it even easier for you to work with.
- Collection view cells are reused for scrolling performance.

Enough **jibber-jabber**. Let's get started.

Create a new **`UICollectionViewController`** subclass called **`BLCImageLibraryViewController`**.

We'll start with the interface, which contains a simple delegate protocol and accompanying property:

BLCImageLibraryViewController.h

```
+ @class BLCImageLibraryViewController;
+
+ @protocol BLCImageLibraryViewControllerDelegate <NSObject>
+
+ - (void) imageLibraryViewController:(BLCImageLibraryViewController *)imageLibraryViewController didCompleteWithImage:(UIImage *)ima
+
+ @end
+
@interface BLCImageLibraryViewController : UICollectionViewController
+
+ @property (nonatomic, weak) NSObject <BLCImageLibraryViewControllerDelegate> *delegate;

@end
```

Let's look at the **`.m`** file. We'll import **`ALAssetsLibrary`** and the crop view controller we just made. We'll also add a few properties and declare

```
#import "BLCImageLibraryViewController.h"
+ #import <AssetsLibrary/AssetsLibrary.h>
+ #import "BLCCropImageViewController.h"

@interface BLCImageLibraryViewController () <BLCCropImageViewControllerDelegate>
+
+ @property (nonatomic, strong) ALAssetsLibrary *library;
+
+ @property (nonatomic, strong) NSMutableArray *groups;
+ @property (nonatomic, strong) NSMutableArray *arraysOfAssets;

@end
```

library represents the user's entire collection of photos and videost.

groups will be an array of **ALAssetsGroup** objects. Each album in the photo library is an **ALAssetsGroup**.

arraysOfAssets will be an array of arrays. Each nested array will contain **ALAsset** objects from the corresponding album.

(**groups** and **arraysOfAssets** will always have the same number of objects.)

Setting Things Up

Because collection views can have any layout imaginable, there's a separate class - **UICollectionViewLayout** - which manages this layout. Apple provides a handy subclass, **UICollectionViewFlowLayout**, which "organizes items into a grid with optional header and footer views for each section."

In the initializer, we'll create this layout, assign an item size (we'll update it later once we know the device's screen size), and initialize our properties:

BLCImageLibraryViewController.m

```
@implementation BLCImageLibraryViewController

+ - (instancetype) init {
+     UICollectionViewFlowLayout *layout = [[UICollectionViewFlowLayout alloc] init];
+     layout.itemSize = CGSizeMake(100, 100);
+
+     self = [super initWithCollectionViewLayout:layout];
+
+     if (self) {
+         self.library = [[ALAssetsLibrary alloc] init];
+         self.groups = [NSMutableArray array];
+         self.arraysOfAssets = [NSMutableArray array];
+     }
+
+     return self;
+ }
```

In **viewDidLoad**, we'll register the default classes for cells and "supplementary views". (Supplementary views are headers and footers.) We als set a background color and make a cancel button.

BLCImageLibraryViewController.m

```
+ {
+     [super viewDidLoad];
+     // Do any additional setup after loading the view.
+
+     [self.collectionView registerClass:[UICollectionViewCell class] forCellWithReuseIdentifier:@"cell"];
+     [self.collectionView registerClass:[UICollectionReusableView class] forSupplementaryViewOfKind:UICollectionElementKindSectionHe
+
+     self.collectionView.backgroundColor = [UIColor whiteColor];
+
+     UIImage *cancelImage = [UIImage imageNamed:@"x"];
+     UIBarButtonItem *cancelButton = [[UIBarButtonItem alloc] initWithImage:cancelImage style:UIBarButtonItemStyleDone target:self a
+     self.navigationItem.leftBarButtonItem = cancelButton;
+ }
+
+ - (void) cancelPressed:(UIBarButtonItem *)sender {
+     [self.delegate imageLibraryViewController:self didCompleteWithImage:nil];
+ }
```

At layout time, we calculate the size of each cell. We want to fit as many as possible on each row, without going below 100 points. We don't pu
any spacing between cells, and we set our header to be 30 points high.

BLCImageLibraryViewController.m

```
+ - (void) viewWillLayoutSubviews {
+     [super viewWillLayoutSubviews];
+
+     CGFloat width = CGRectGetWidth(self.view.frame);
+     CGFloat minWidth = 100;
+     NSInteger divisor = width / minWidth;
+     CGFloat cellSize = width / divisor;
+
+     UICollectionViewFlowLayout *flowLayout = (UICollectionViewFlowLayout *)self.collectionViewLayout;
+     flowLayout.itemSize = CGSizeMake(cellSize, cellSize);
+     flowLayout.minimumInteritemSpacing = 0;
+     flowLayout.minimumLineSpacing = 0;
+     flowLayout.headerReferenceSize = CGSizeMake(width, 30);
+ }
```

Displaying the Images

In **`viewWillAppear:`**, we ask the library to enumerate through all of the groups (**`ALAssetsGroup`**).

For each group, we add the group to **`self.groups`**.

We then enumerate through each asset (**`ALAsset`**) in each group and add them to the array. These are either images or videos.

If the request to enumerate groups fails for some reason (usually a permissions issue), we display an alert to the user:

BLCImageLibraryViewController.m

```
+     [super viewWillAppear:animated];
+
+     [self.groups removeAllObjects];
+     [self.arraysOfAssets removeAllObjects];
+
+     [self.library enumerateGroupsWithTypes:ALAssetsGroupSavedPhotos | ALAssetsGroupAlbum usingBlock:^(ALAssetsGroup *group, BOOL *s
+         if (group) {
+             [self.groups addObject:group];
+             NSMutableArray *assets = [NSMutableArray array];
+             [self.arraysOfAssets addObject:assets];
+
+             [group enumerateAssetsUsingBlock:^(ALAsset *result, NSUInteger index, BOOL *stop) {
+                 if (result) {
+                     [assets addObject:result];
+                 }
+             }];
+
+             [self.collectionView reloadData];
+         }
+     } failureBlock:^(NSError *error) {
+         UIAlertView *alert = [[UIAlertView alloc] initWithTitle:error.localizedDescription message:error.localizedRecoverySuggestio
+         [alert show];
+
+         [self.collectionView reloadData];
+     }];
+ }
```

One thing you might notice is that this is the **only** use of **`self.library`** aside from when we initialized it. The reason we have to keep it as a property is that the objects vended by the library (the group and asset objects) will stop giving out information if the library is deallocated. Thi is **explained in the class reference**:

> *The lifetimes of objects you get back from a library instance are tied to the lifetime of the library instance.*

The reason for this is so that the objects can stop providing data if the user disables permissions.

To save memory, we'll also clear this images off the screen when they're no longer being displayed:

BLCImageLibraryViewController.m

```
+ - (void) viewWillDisappear:(BOOL)animated {
+     [super viewWillDisappear:animated];
+
+     [self.groups removeAllObjects];
+     [self.arraysOfAssets removeAllObjects];
+     [self.collectionView reloadData];
+ }
```

Implementing Collection View Delegate and Data Source Methods

Just like with **`UITableView`**, we need to tell the collection view how many sections there are:

BLCImageLibraryViewController.m

```
+ - (NSInteger) numberOfSectionsInCollectionView:(UICollectionView *)collectionView {
+     return self.groups.count;
+ }
```

We also need to tell it how many items are in each section:

BLCImageLibraryViewController.m

```
+     NSArray *imagesArray = self.arraysOfAssets[section];
+
+     if (imagesArray) {
+         return imagesArray.count;
+     }
+
+     return 0;
+ }
```

And, given a section and a row (in the form of an **`NSIndexPath`**), we need to produce a **`UICollectionViewCell`** for the collection view to displa on the screen.

Ours will simply have one image view, which takes up the entire cell. We'll place a rendering of the asset's **`thumbnail`** inside:

BLCImageLibraryViewController.m

```
+ - (UICollectionViewCell *) collectionView:(UICollectionView *)collectionView cellForItemAtIndexPath:(NSIndexPath *)indexPath {
+     static NSInteger imageViewTag = 54321;
+
+     UICollectionViewCell *cell = [collectionView dequeueReusableCellWithReuseIdentifier:@"cell" forIndexPath:indexPath];
+
+     UIImageView *imageView = (UIImageView *)[cell.contentView viewWithTag:imageViewTag];
+
+     if (!imageView) {
+         imageView = [[UIImageView alloc] initWithFrame:cell.contentView.bounds];
+         imageView.tag = imageViewTag;
+         imageView.autoresizingMask = UIViewAutoresizingFlexibleHeight | UIViewAutoresizingFlexibleWidth;
+         imageView.contentMode = UIViewContentModeScaleAspectFill;
+         imageView.clipsToBounds = YES;
+         [cell.contentView addSubview:imageView];
+     }
+
+     ALAsset *asset = self.arraysOfAssets[indexPath.section][indexPath.row];
+     CGImageRef imageRef = asset.thumbnail;
+
+     UIImage *image;
+
+     if (imageRef) {
+         image = [UIImage imageWithCGImage:imageRef];
+     }
+
+     imageView.image = image;
+
+     return cell;
+ }
```

Similarly, given a section, we need need to provide a **`UICollectionReusableView`** that represents a section header. This code is nearly identical, just with **`UILabel`** instead of **`UIImageView`**:

BLCImageLibraryViewController.m

```
+     UICollectionReusableView *view = [collectionView dequeueReusableSupplementaryViewOfKind:kind withReuseIdentifier:@"reusable vie
+
+     if ([kind isEqual:UICollectionElementKindSectionHeader]) {
+         static NSInteger headerLabelTag = 2468;
+
+         UILabel *label = (UILabel *)[view viewWithTag:headerLabelTag];
+
+         if (!label) {
+             label = [[UILabel alloc] initWithFrame:view.bounds];
+             label.tag = headerLabelTag;
+             label.autoresizingMask = UIViewAutoresizingFlexibleHeight | UIViewAutoresizingFlexibleWidth;
+             label.textAlignment = NSTextAlignmentCenter;
+
+             label.backgroundColor = [UIColor colorWithRed:230/255.0f green:230/255.0f blue:235/255.0f alpha:1.0f];
+
+             [view addSubview:label];
+         }
+
+         ALAssetsGroup *group = self.groups[indexPath.section];
+
+         //Use any color you want or skip defining it
+         UIColor* textColor = [UIColor colorWithWhite:0.35 alpha:1];
+
+         NSDictionary *textAttributes = @{NSForegroundColorAttributeName : textColor,
+                                          NSFontAttributeName : [UIFont fontWithName:@"HelveticaNeue-Medium" size:14],
+                                          NSTextEffectAttributeName : NSTextEffectLetterpressStyle};
+
+         NSAttributedString* attributedString;
+
+
+         NSString *groupName = [group valueForProperty:ALAssetsGroupPropertyName];
+
+         if (groupName) {
+             attributedString = [[NSAttributedString alloc] initWithString:groupName attributes:textAttributes];
+         }
+
+         label.attributedText = attributedString;
+     }
+
+     return view;
+ }
```

> ***Common Bug Warning.*** **`NSAttributedString`** *will throw an exception if you pass it a* **`nil`** *string. Always check that your string isn't* **`nil`** *before passing it to* **`NSAttributedString`***'s initializer.*

One thing that you haven't seen before here is setting **`NSTextEffectAttributeName`** to **`NSTextEffectLetterpressStyle`**, which will apply a beautiful letterpress effect to the text.

Responding to Image Selection

When the user taps a thumbnail, let's get the full resolution image, and pass it to the new crop controller we made:

BLCImageLibraryViewController.m

```
+     ALAsset *asset = self.arraysOfAssets[indexPath.section][indexPath.row];
+     ALAssetRepresentation *representation = asset.defaultRepresentation;
+     CGImageRef imageRef = representation.fullResolutionImage;
+
+     UIImage *imageToCrop;
+
+     if (imageRef) {
+         imageToCrop = [UIImage imageWithCGImage:imageRef scale:representation.scale orientation:(UIImageOrientation)representation.
+     }
+
+     BLCCropImageViewController *cropVC = [[BLCCropImageViewController alloc] initWithImage:imageToCrop];
+     cropVC.delegate = self;
+     [self.navigationController pushViewController:cropVC animated:YES];
+ }
```

And if the user crops an image, let's inform the image library controller's delegate:

BLCImageLibraryViewController.m

```
+ #pragma mark - BLCCropImageViewControllerDelegate
+
+ - (void) cropControllerFinishedWithImage:(UIImage *)croppedImage {
+     [self.delegate imageLibraryViewController:self didCompleteWithImage:croppedImage];
+ }
```

Adding the Images Library Controller

We have two places to add the images library controller:

1. In the camera controller, if the user taps the right button in the camera toolbar
2. In the primary view, if the user taps the camera button but doesn't have a camera available

In **`BLCCameraViewController`**, import the image library controller and declare that it conforms to the delegate protocol:

BLCCameraViewController.m

```
 #import "BLCCropBox.h"
+ #import "BLCImageLibraryViewController.h"

- @interface BLCCameraViewController () <BLCCameraToolbarDelegate, UIAlertViewDelegate>
+ @interface BLCCameraViewController () <BLCCameraToolbarDelegate, UIAlertViewDelegate, BLCImageLibraryViewControllerDelegate>
```

Push the view controller on the navigation stack when the user presses the right button:

BLCCameraViewController.m

```
 - (void) rightButtonPressedOnToolbar:(BLCCameraToolbar *)toolbar {
-     NSLog(@"Photo library button pressed.");
+     BLCImageLibraryViewController *imageLibraryVC = [[BLCImageLibraryViewController alloc] init];
+     imageLibraryVC.delegate = self;
+     [self.navigationController pushViewController:imageLibraryVC animated:YES];
 }
```

When the image library controller hands us an image, pass it to the camera controller's delegate:

BLCCameraViewController.m

```
+ #pragma mark - BLCImageLibraryViewControllerDelegate
+
+ - (void) imageLibraryViewController:(BLCImageLibraryViewController *)imageLibraryViewController didCompleteWithImage:(UIImage *)ima
+     [self.delegate cameraViewController:self didCompleteWithImage:image];
+ }
```

In **`BLCImagesTableViewController`**, we'll take a similar approach.

Update the imports and protocol conformity declaration:

BLCImagesTableViewController.m

```
#import "BLCMediaFullScreenAnimator.h"
#import "BLCCameraViewController.h"
+ #import "BLCImageLibraryViewController.h"

- @interface BLCImagesTableViewController () <BLCMediaTableViewCellDelegate, UIViewControllerTransitioningDelegate, BLCCameraViewCont
+ @interface BLCImagesTableViewController () <BLCMediaTableViewCellDelegate, UIViewControllerTransitioningDelegate, BLCCameraViewCont
```

Update the pragma mark, using the **Oxford comma** to avoid confusion:

BLCImagesTableViewController.m

```
- #pragma mark - Camera and BLCCameraViewControllerDelegate
+ #pragma mark - Camera, BLCCameraViewControllerDelegate, and BLCImageLibraryViewControllerDelegate
```

Update **`cameraPressed:`** to display the different controller if the camera's unavailable:

BLCImagesTableViewController.m

```
 - (void) cameraPressed:(UIBarButtonItem *) sender {
+
+     UIViewController *imageVC;
+
+     if ([UIImagePickerController isSourceTypeAvailable:UIImagePickerControllerSourceTypeCamera]) {
         BLCCameraViewController *cameraVC = [[BLCCameraViewController alloc] init];
         cameraVC.delegate = self;
-     UINavigationController *nav = [[UINavigationController alloc] initWithRootViewController:cameraVC];
+         imageVC = cameraVC;
+     } else if ([UIImagePickerController isSourceTypeAvailable:UIImagePickerControllerSourceTypeSavedPhotosAlbum]) {
+         BLCImageLibraryViewController *imageLibraryVC = [[BLCImageLibraryViewController alloc] init];
+         imageLibraryVC.delegate = self;
+         imageVC = imageLibraryVC;
+     }
+
+     if (imageVC) {
+         UINavigationController *nav = [[UINavigationController alloc] initWithRootViewController:imageVC];
         [self presentViewController:nav animated:YES completion:nil];
+     }

     return;
 }
```

Implement the delegate method:

BLCImagesTableViewController.m

```
+ - (void) imageLibraryViewController:(BLCImageLibraryViewController *)imageLibraryViewController didCompleteWithImage:(UIImage *)ima
+     [imageLibraryViewController dismissViewControllerAnimated:YES completion:^{
+         if (image) {
+             NSLog(@"Got an image!");
+         } else {
+             NSLog(@"Closed without an image.");
+         }
+     }];
+ }
```

Run the app. You can now take pictures, and also select existing pictures from your library.

In the next checkpoint, we'll allow the users to send their pictures to the Instagram app for posting.

- To give your image library a more unique look, play with some of the properties and calculations in `[BLCImageLibraryViewController -viewWillLayoutSubviews]`. Experiment with different sizes and padding.
- Try giving each cell a *different* size by conforming to the `UICollectionViewDelegateFlowLayout` protocol and implementing `collectionView:layout:sizeForItemAtIndexPath:`.
- The camera crop view looks better than the image library's crop view because of the two translucent `UIToolbar` objects. Add a similar effect to `BLCCropViewController`. You can accomplish this by making similar toolbars in this controller, or by moving them into `BLCCropBox`.

When you're satisfied with your project and the assignment, commit, merge and push your code to GitHub:

Terminal

```
$ git add .
$ git commit -m 'Added Photo Library and Crop View'
$ git checkout master
$ git merge using-assets-library
$ git push
```

Submit this checkpoint's assignment with links for your repo and commit.

assignment completed

MADE BY BLOC

Tech Talks & Resources

Programming Bootcamp Comparison

Jottly: A Beginner's Guide to HTML, CSS, Skeleton and Animate.css

Swiftris: Build Your First iOS Game with Swift

Webflow Tutorial: Design Responsive Sites with Webflow

Ruby Warrior

Bloc's Diversity Scholarship

SIGN UP FOR OUR MAILING LIST

Send

hello@bloc.io

Considering enrolling? (404) 480-2562

Partnership / Corporate Inquiries? (650) 741-5682

Adding Cool Filters and Sending to Instagram

In this checkpoint, you'll learn three new things:

1. How to use *core image filters*, and how to combine them your own to create custom effects
2. How to use *operation queues* (**`NSOperationQueue`**) to handle batches of long-running work
3. How to use a *document interaction controller** to send documents to other apps (we'll send the cropped and filtered photo to the Instagram iOS app for posting)

`cd` into your Blocstagram directory and make a new **`git`** branch:

Terminal

```
$ git checkout -b filters-and-posting
```

Create a View Controller for Filtering and Posting

Create new **`UIViewController`** subclass called **`BLCPostToInstagramViewController`**.

The public interface in the **`.h`** file will be simple:

BLCPostToInstagramViewController.h

```
#import <UIKit/UIKit.h>

@interface BLCPostToInstagramViewController : UIViewController

+ - (instancetype) initWithImage:(UIImage *)sourceImage;
+
@end
```

Before we go any further, let's update **`BLCImagesTableViewController`** to use this class.

Import the file:

BLCImagesTableViewController.m

```
#import "BLCCameraViewController.h"
#import "BLCImageLibraryViewController.h"
+ #import "BLCPostToInstagramViewController.h"
```

Create a new method to push the controller, and call it from both respective delegate methods:

BLCImagesTableViewController.m

```
+ - (void) handleImage:(UIImage *)image withNavigationController:(UINavigationController *)nav {
+     if (image) {
+         BLCPostToInstagramViewController *postVC = [[BLCPostToInstagramViewController alloc] initWithImage:image];
+
+         [nav pushViewController:postVC animated:YES];
+     } else {
+         [nav dismissViewControllerAnimated:YES completion:nil];
+     }
+ }

 - (void) cameraViewController:(BLCCameraViewController *)cameraViewController didCompleteWithImage:(UIImage *)image {
-     [cameraViewController dismissViewControllerAnimated:YES completion:^{
-         if (image) {
-             NSLog(@"Got an image!");
-         } else {
-             NSLog(@"Closed without an image.");
-         }
-     }];
+     [self handleImage:image withNavigationController:cameraViewController.navigationController];
 }

 - (void) imageLibraryViewController:(BLCImageLibraryViewController *)imageLibraryViewController didCompleteWithImage:(UIImage *)image
-     [imageLibraryViewController dismissViewControllerAnimated:YES completion:^{
-         if (image) {
-             NSLog(@"Got an image!");
-         } else {
-             NSLog(@"Closed without an image.");
-         }
-     }];
+     [self handleImage:image withNavigationController:imageLibraryViewController.navigationController];
 }
```

In the **`.m`** file, declare that we conform to the collection view delegate and data source protocols, as well as the **`UIAlertViewDelegate`** and **`UIDocumentInteractionControllerDelegate`** protocols.

We'll also need a few properties:

```
#import "BLCPostToInstagramViewController.h"

- @interface BLCPostToInstagramViewController ()
+ @interface BLCPostToInstagramViewController () <UICollectionViewDataSource, UICollectionViewDelegate, UIAlertViewDelegate, UIDocume

+ @property (nonatomic, strong) UIImage *sourceImage;
+ @property (nonatomic, strong) UIImageView *previewImageView;
+
+ @property (nonatomic, strong) NSOperationQueue *photoFilterOperationQueue;
+ @property (nonatomic, strong) UICollectionView *filterCollectionView;
+
+ @property (nonatomic, strong) NSMutableArray *filterImages;
+ @property (nonatomic, strong) NSMutableArray *filterTitles;
+
+ @property (nonatomic, strong) UIButton *sendButton;
+ @property (nonatomic, strong) UIBarButtonItem *sendBarButton;
+
@end
```

Here's what the properties will do:

- **sourceImage** will store the image passed into **initWithImage:**
- **previewImageView** will display the image with its current filter
- **photoFilterOperationQueue** will store our photo filter operations; more on that later
- **filterCollectionView** will be a collection view that shows all of the filters available
- **filterImages** and **filterTitles** will hold filtered images and their titles
- **sendButton** will be the big purple "send to Instagram" button
- **sendBarButton** will show on short iPhones in the navigation bar where there's no room for **sendButton**

Let's start by writing the initializer:

BLCPostToInstagramViewController.m

```
+ @implementation BLCPostToInstagramViewController
+
+ - (instancetype) initWithImage:(UIImage *)sourceImage {
+     self = [super init];
+
+     if (self) {
+         self.sourceImage = sourceImage;
+         self.previewImageView = [[UIImageView alloc] initWithImage:self.sourceImage];
+
+         self.photoFilterOperationQueue = [[NSOperationQueue alloc] init];
+
+         UICollectionViewFlowLayout *flowLayout = [[UICollectionViewFlowLayout alloc] init];
+         flowLayout.itemSize = CGSizeMake(44, 64);
+         flowLayout.scrollDirection = UICollectionViewScrollDirectionHorizontal;
+         flowLayout.minimumInteritemSpacing = 10;
+         flowLayout.minimumLineSpacing = 10;
+
+         self.filterCollectionView = [[UICollectionView alloc] initWithFrame:CGRectZero collectionViewLayout:flowLayout];
+         self.filterCollectionView.dataSource = self;
+         self.filterCollectionView.delegate = self;
+         self.filterCollectionView.showsHorizontalScrollIndicator = NO;
+
+         self.filterImages = [NSMutableArray arrayWithObject:sourceImage];
+         self.filterTitles = [NSMutableArray arrayWithObject:NSLocalizedString(@"None", @"Label for when no filter is applied to a p
+
+         self.sendButton = [UIButton buttonWithType:UIButtonTypeSystem];
+         self.sendButton.backgroundColor = [UIColor colorWithRed:0.345 green:0.318 blue:0.424 alpha:1]; /*#58516c*/
+         self.sendButton.layer.cornerRadius = 5;
+         [self.sendButton addTarget:self action:@selector(sendButtonPressed:) forControlEvents:UIControlEventTouchUpInside];
+
+         self.sendBarButton = [[UIBarButtonItem alloc] initWithTitle:NSLocalizedString(@"Send", @"Send button") style:UIBarButtonIte
+     }
+
+     return self;
+ }
```

This should look pretty familiar for the most part, but let's walk through it. We store the source image passed in and initialize the preview image view with that image. We create the operation queue. We create and configure a **`UICollectionViewFlowLayout`** instance to define the layout of our filter collection view, and then use it to initialize a **`UICollectionView`**. We set the collection view's **`delegate`** and **`dataSource`** properties. (We don't need to assign these properties in a **`UICollectionViewController`**, but we're subclassing plain old **`UIViewController`**.) We disable **`showsHorizontalScrollIndicator`** to hide the horizontal scroll indicator; this is just an aesthetic preference.

We initialize the **`filterImages`** and **`filterTitles`** arrays. The first object in each array represents the unfiltered image. (This will allow a user to remove a filter once it's applied.) Finally, we create both **`sendButton`** and **`sendBarButton`**; note that they both have the same target-action method.

We'll need to add text to the button, so add this to **`initWithImage:`**:

BLCPostToInstagramViewController.m

```
        self.sendButton.layer.cornerRadius = 5;
+         [self.sendButton setAttributedTitle:[self sendAttributedString] forState:UIControlStateNormal];
        [self.sendButton addTarget:self action:@selector(sendButtonPressed:) forControlEvents:UIControlEventTouchUpInside];
```

Add this method to configure the text:

BLCPostToInstagramViewController.m

```
+ #pragma mark - Buttons
+
+ - (NSAttributedString *) sendAttributedString {
+     NSString *baseString = NSLocalizedString(@"SEND TO INSTAGRAM", @"send to Instagram button text");
+     NSRange range = [baseString rangeOfString:baseString];
+
+     NSMutableAttributedString *commentString = [[NSMutableAttributedString alloc] initWithString:baseString];
+
+     [commentString addAttribute:NSFontAttributeName value:[UIFont fontWithName:@"HelveticaNeue-Bold" size:13] range:range];
+     [commentString addAttribute:NSKernAttributeName value:@1.3 range:range];
+     [commentString addAttribute:NSForegroundColorAttributeName value:[UIColor colorWithRed:0.933 green:0.933 blue:0.933 alpha:1] ra
+
+     return commentString;
+ }
```

In **`viewDidLoad`**, we'll configure the view, add subviews to the view hierarchy, and decide which button to use:

BLCPostToInstagramViewController.m

```
 - (void)viewDidLoad
 {
     [super viewDidLoad];
     // Do any additional setup after loading the view.
+
+     [self.view addSubview:self.previewImageView];
+     [self.view addSubview:self.filterCollectionView];
+
+     if (CGRectGetHeight(self.view.frame) > 500) {
+         [self.view addSubview:self.sendButton];
+     } else {
+         self.navigationItem.rightBarButtonItem = self.sendBarButton;
+     }
+
+     [self.filterCollectionView registerClass:[UICollectionViewCell class] forCellWithReuseIdentifier:@"cell"];
+
+     self.view.backgroundColor = [UIColor whiteColor];
+     self.filterCollectionView.backgroundColor = [UIColor whiteColor];
+
+     self.navigationItem.title = NSLocalizedString(@"Apply Filter", @"apply filter view title");
 }
```

Note the class of the cell we're registering. Just like in the prior checkpoint, we'll use a vanilla **`UICollectionViewCell`** instead of subclassing it

In **viewWillLayoutSubviews** we'll position the views:

- If the view's height is **>** **500**, we'll include **sendButton** and allow the collection view to take up whatever height remains
- If the view's height is **<** **500**, the send button is hidden and the collection view will go to the bottom of the view.

We'll also update the **UICollectionViewFlowLayout** with the correct size based on the collection view's determined height.

Here's how it looks:

BLCPostToInstagramViewController.m

```
+ - (void)viewWillLayoutSubviews {
+     [super viewWillLayoutSubviews];
+
+     CGFloat edgeSize = MIN(CGRectGetWidth(self.view.frame), CGRectGetHeight(self.view.frame));
+
+     self.previewImageView.frame = CGRectMake(0, self.topLayoutGuide.length, edgeSize, edgeSize);
+
+     CGFloat buttonHeight = 50;
+     CGFloat buffer = 10;
+
+     CGFloat filterViewYOrigin = CGRectGetMaxY(self.previewImageView.frame) + buffer;
+     CGFloat filterViewHeight;
+
+     if (CGRectGetHeight(self.view.frame) > 500) {
+         self.sendButton.frame = CGRectMake(buffer, CGRectGetHeight(self.view.frame) - buffer - buttonHeight, CGRectGetWidth(self.vi
+
+         filterViewHeight = CGRectGetHeight(self.view.frame) - filterViewYOrigin - buffer - buffer - CGRectGetHeight(self.sendButton
+     } else {
+         filterViewHeight = CGRectGetHeight(self.view.frame) - CGRectGetMaxY(self.previewImageView.frame) - buffer - buffer;
+     }
+
+     self.filterCollectionView.frame = CGRectMake(0, filterViewYOrigin, CGRectGetWidth(self.view.frame), filterViewHeight);
+
+     UICollectionViewFlowLayout *flowLayout = (UICollectionViewFlowLayout *)self.filterCollectionView.collectionViewLayout;
+     flowLayout.itemSize = CGSizeMake(CGRectGetHeight(self.filterCollectionView.frame) - 20, CGRectGetHeight(self.filterCollectionVi
+ }
```

Configuring the Collection View

The collection view will always have only 1 section:

BLCPostToInstagramViewController.m

```
+ #pragma mark - UICollectionView delegate and data source
+
+ - (NSInteger) numberOfSectionsInCollectionView:(UICollectionView *)collectionView {
+     return 1;
+ }
```

The number of items will always be equal to the number of filtered images available:

BLCPostToInstagramViewController.m

```
+ - (NSInteger) collectionView:(UICollectionView *)collectionView numberOfItemsInSection:(NSInteger)section {
+     return self.filterImages.count;
+ }
```

When the cell loads, we'll make sure there's an image view and a label on it, and we'll set their content from the appropriate arrays:

BLCPostToInstagramViewController.m

```
+     UICollectionViewCell *cell = [collectionView dequeueReusableCellWithReuseIdentifier:@"cell" forIndexPath:indexPath];
+
+     static NSInteger imageViewTag = 1000;
+     static NSInteger labelTag = 1001;
+
+     UIImageView *thumbnail = (UIImageView *)[cell.contentView viewWithTag:imageViewTag];
+     UILabel *label = (UILabel *)[cell.contentView viewWithTag:labelTag];
+
+     UICollectionViewFlowLayout *flowLayout = (UICollectionViewFlowLayout *)self.filterCollectionView.collectionViewLayout;
+     CGFloat thumbnailEdgeSize = flowLayout.itemSize.width;
+
+     if (!thumbnail) {
+         thumbnail = [[UIImageView alloc] initWithFrame:CGRectMake(0, 0, thumbnailEdgeSize, thumbnailEdgeSize)];
+         thumbnail.contentMode = UIViewContentModeScaleAspectFill;
+         thumbnail.tag = imageViewTag;
+         thumbnail.clipsToBounds = YES;
+
+         [cell.contentView addSubview:thumbnail];
+     }
+
+     if (!label) {
+         label = [[UILabel alloc] initWithFrame:CGRectMake(0, thumbnailEdgeSize, thumbnailEdgeSize, 20)];
+         label.tag = labelTag;
+         label.textAlignment = NSTextAlignmentCenter;
+         label.font = [UIFont fontWithName:@"HelveticaNeue-Medium" size:10];
+         [cell.contentView addSubview:label];
+     }
+
+     thumbnail.image = self.filterImages[indexPath.row];
+     label.text = self.filterTitles[indexPath.row];
+
+     return cell;
+ }
```

We set the frames of these items based on the flow layout's **`itemSize`** property, which we set earlier in **`viewWillLayoutSubviews`**.

If the user taps on one of the cells, we'll update the preview image to show the image with that filter:

BLCPostToInstagramViewController.m

```
+ - (void) collectionView:(UICollectionView *)collectionView didSelectItemAtIndexPath:(NSIndexPath *)indexPath {
+     self.previewImageView.image = self.filterImages[indexPath.row];
+ }
```

Working with Photo Filters

When working with photo filters, we need to use a few new classes:

1. **`CIFilter`** represents one photo filter. It takes one or more inputs (for example, images, colors, angles, etc.)
2. **`CIImage`** represents an image within Core Image. A **`CIImage`** isn't fully drawn, allowing filters to work faster.
3. **`CIVector`** represents a series of numbers, typically representing positions, areas, or color values.

You can see a **list of all available filters**. (Note that some require specific versions of iOS, and some are only available on OS X.)

The **`CIPhotoEffectNoir`** filter looks pretty good, so let's start with that:

Here's how you would generally create the filtered image from a **`UIImage`**:

```
UIImage *sourceImage = // some image
CIImage *sourceCIImage = [CIImage imageWithCGImage:sourceImage];
CIFilter *noirFilter = [CIFilter filterWithName:@"CIPhotoEffectNoir"];

if (noirFilter) {
    [noirFilter setValue:sourceCIImage forKey:kCIInputImageKey];
    CIImage *outputCIImage = noirFilter.outputImage;
    UIImage *outputImage = [UIImage imageWithCIImage:outputCIImage scale:self.sourceImage.scale orientation:self.sourceImage.imageOrie

    if (outputImage) {
        // do something with outputImage
    }
}
```

Here's a few quirks you should know about **`CIFilter`**:

- For specialized behavior within most objects you'd create a *subclass*. With **`CIFilter`**, you instead call **`filterWithName:`** and specify the filter you want as an **`NSString`**.
- Because different filters have different settings, but they all share the same class (**`CIFilter`**), you don't use properties to control a **`CIFilter`**'s settings. Instead, you call **`setValue:forKey:`** on the filter. (A list of keys is in each filter's documentation.)
- Just like **`NSURL`**, you should always check to make sure it's not **`nil`** before you use it.

Between the image conversions and the filter itself, this code can take a few seconds to run. We don't want to run it on the main thread, and since we have a lot of filters, it could get annoying to keep track of everything using only **`dispatch_async`**. We'll use a class called **`NSOperationQueue`** that makes it very easy to schedule pieces of long-running code (or "operations", as they're called.)

First, make a method to handle finished filters and add them to the collection view:

BLCPostToInstagramViewController.m

```
+
+ - (void) addCIImageToCollectionView:(CIImage *)CIImage withFilterTitle:(NSString *)filterTitle {
+     UIImage *image = [UIImage imageWithCIImage:CIImage scale:self.sourceImage.scale orientation:self.sourceImage.imageOrientation];
+
+     if (image) {
+         // Decompress image
+         UIGraphicsBeginImageContextWithOptions(image.size, NO, image.scale);
+         [image drawAtPoint:CGPointZero];
+         image = UIGraphicsGetImageFromCurrentImageContext();
+         UIGraphicsEndImageContext();
+
+         dispatch_async(dispatch_get_main_queue(), ^{
+             NSUInteger newIndex = self.filterImages.count;
+
+             [self.filterImages addObject:image];
+             [self.filterTitles addObject:filterTitle];
+
+             [self.filterCollectionView insertItemsAtIndexPaths:@[[NSIndexPath indexPathForItem:newIndex inSection:0]]];
+         });
+     }
+ }
```

This code does the following:

1. Converts the **`CIImage`** to a **`UIImage`**. Because **`CIImage`** isn't fully rendered, the output **`UIImage`** is slow to draw.
2. Forces the **`UIImage`** to draw, and then saves the drawn `UIImage.
3. On the main thread, adds the completed **`UIImage`** and filter title to the arrays, and tells the collection view that a new item is available.

Now we just need to add a filter to the operation queue. Create a new method:

BLCPostToInstagramViewController.m

```
+ - (void) addFiltersToQueue {
+     CIImage *sourceCIImage = [CIImage imageWithCGImage:self.sourceImage.CGImage];
+
+     // Noir filter
+
+     [self.photoFilterOperationQueue addOperationWithBlock:^{
+         CIFilter *noirFilter = [CIFilter filterWithName:@"CIPhotoEffectNoir"];
+
+         if (noirFilter) {
+             [noirFilter setValue:sourceCIImage forKey:kCIInputImageKey];
+             [self addCIImageToCollectionView:noirFilter.outputImage withFilterTitle:NSLocalizedString(@"Noir", @"Noir Filter")];
+         }
+     }];
+ }
```

`addOperationWithBlock:` takes a block of code and adds it to the operation queue, which means it will run eventually.

> *The operation queue executes as many blocks as it can, up to its* **`maxConcurrentOperationCount`**. *The default value for this is set based on the hardware the user has. If you're interested in experimenting, try setting this property to* **`1`** *and the filters will come in slower.*

Because more than 1 operation can run at a time, the operations won't necessarily finish in the same order they're started. You may see filter appear in a different order each time you run the app.

Add a call to **`addFiltersToQueue`** to the end of **`initWithImage:`**:

BLCPostToInstagramViewController.m

```
+          [self addFiltersToQueue];
     }

     return self;
 }
```

Run the app and try to attach a photo. You should now be able to switch between the **None** and **Noir** filters.

Let's add some more filters. The code to implement them will look nearly identical:

BLCPostToInstagramViewController.m

```
             [self addCIImageToCollectionView:noirFilter.outputImage withFilterTitle:NSLocalizedString(@"Noir", @"Noir Filter")];
         }
     }];
 }
+
+    // Boom filter
+
+    [self.photoFilterOperationQueue addOperationWithBlock:^{
+        CIFilter *boomFilter = [CIFilter filterWithName:@"CIPhotoEffectProcess"];
+
+        if (boomFilter) {
+            [boomFilter setValue:sourceCIImage forKey:kCIInputImageKey];
+            [self addCIImageToCollectionView:boomFilter.outputImage withFilterTitle:NSLocalizedString(@"Boom", @"Boom Filter")];
+        }
+    }];
+
+    // Warm filter
+
+    [self.photoFilterOperationQueue addOperationWithBlock:^{
+        CIFilter *warmFilter = [CIFilter filterWithName:@"CIPhotoEffectTransfer"];
+
+        if (warmFilter) {
+            [warmFilter setValue:sourceCIImage forKey:kCIInputImageKey];
+            [self addCIImageToCollectionView:warmFilter.outputImage withFilterTitle:NSLocalizedString(@"Warm", @"Warm Filter")];
+        }
+    }];
+
+    // Pixel filter
+
+    [self.photoFilterOperationQueue addOperationWithBlock:^{
+        CIFilter *pixelFilter = [CIFilter filterWithName:@"CIPixellate"];
+
+        if (pixelFilter) {
+            [pixelFilter setValue:sourceCIImage forKey:kCIInputImageKey];
+            [self addCIImageToCollectionView:pixelFilter.outputImage withFilterTitle:NSLocalizedString(@"Pixel", @"Pixel Filter")];
+        }
+    }];
+
+    // Moody filter
+
+    [self.photoFilterOperationQueue addOperationWithBlock:^{
+        CIFilter *moodyFilter = [CIFilter filterWithName:@"CISRGBToneCurveToLinear"];
+
+        if (moodyFilter) {
+            [moodyFilter setValue:sourceCIImage forKey:kCIInputImageKey];
+            [self addCIImageToCollectionView:moodyFilter.outputImage withFilterTitle:NSLocalizedString(@"Moody", @"Moody Filter")];
+        }
+    }];
 }
```

These filters all work nearly identically; we're just passing in a different value to **`filterWithName:`**.

Let's take a look at a slightly more complex filter, which we'll call **Drunk**. This filter will use the **`CIStraightenFilter`**, which is usually used for straightening, to tilt the image 0.2 radians (about 11.5 degrees). We'll combine this filter with the **`CIConvolution5X5`** filter to create a blurry double-vision effect.

BLCPostToInstagramViewController.m

```
            [self addCIImageToCollectionView:moodyFilter.outputImage withFilterTitle:NSLocalizedString(@"Moody", @"Moody Filter")];
        }
    }];
+
+     // Drunk filter
+
+     [self.photoFilterOperationQueue addOperationWithBlock:^{
+         CIFilter *drunkFilter = [CIFilter filterWithName:@"CIConvolution5X5"];
+         CIFilter *tiltFilter = [CIFilter filterWithName:@"CIStraightenFilter"];
+
+         if (drunkFilter) {
+             [drunkFilter setValue:sourceCIImage forKey:kCIInputImageKey];
+
+             CIVector *drunkVector = [CIVector vectorWithString:@"[0.5 0 0 0 0 0 0 0 0 0.05 0 0 0 0 0 0 0 0 0 0 0.05 0 0 0 0.5]"];
+             [drunkFilter setValue:drunkVector forKeyPath:@"inputWeights"];
+
+             CIImage *result = drunkFilter.outputImage;
+
+             if (tiltFilter) {
+                 [tiltFilter setValue:result forKeyPath:kCIInputImageKey];
+                 [tiltFilter setValue:@0.2 forKeyPath:kCIInputAngleKey];
+                 result = tiltFilter.outputImage;
+             }
+
+             [self addCIImageToCollectionView:result withFilterTitle:NSLocalizedString(@"Drunk", @"Drunk Filter")];
+         }
+     }];
 }
```

The drunk filter takes a **`CIVector`** which specifies a 5x5 matrix convolution. (Matrix image convolution is a somewhat complex topic outside the scope of this checkpoint, but if you're interested in learning about it, check out **this Image Convolution presentation from a Portland State University class**.)

The **`outputImage`** of the first filter is then passed into the **`CIStraightenFilter`**. We also pass in the **`NSNumber 0.2`** for the **`kCIInputAngleKey`**, which is specified in radians.

Note that we don't need to convert the **`CIImage`** to a **`UIImage`** and then back - you can easily pass a **`CIImage`** around between many filters before it's completed. In fact, let's write an even more extreme example, where we'll combine lots of filters to create the look of old film.

Here's the code; we'll step through it on the other side:

BLCPostToInstagramViewController.m

```
        }
    }];
+
+    // Film filter
+
+    [self.photoFilterOperationQueue addOperationWithBlock:^{
+        CIFilter *sepiaFilter = [CIFilter filterWithName:@"CISepiaTone"];
+        [sepiaFilter setValue:@1 forKey:kCIInputIntensityKey];
+        [sepiaFilter setValue:sourceCIImage forKey:kCIInputImageKey];
+
+        CIFilter *randomFilter = [CIFilter filterWithName:@"CIRandomGenerator"];
+
+        CIImage *randomImage = [CIFilter filterWithName:@"CIRandomGenerator"].outputImage;
+        CIImage *otherRandomImage = [randomImage imageByApplyingTransform:CGAffineTransformMakeScale(1.5, 25.0)];
+
+        CIFilter *whiteSpecks = [CIFilter filterWithName:@"CIColorMatrix" keysAndValues:kCIInputImageKey, randomImage,
+                                 @"inputRVector", [CIVector vectorWithX:0.0 Y:1.0 Z:0.0 W:0.0],
+                                 @"inputGVector", [CIVector vectorWithX:0.0 Y:1.0 Z:0.0 W:0.0],
+                                 @"inputBVector", [CIVector vectorWithX:0.0 Y:1.0 Z:0.0 W:0.0],
+                                 @"inputAVector", [CIVector vectorWithX:0.0 Y:0.01 Z:0.0 W:0.0],
+                                 @"inputBiasVector", [CIVector vectorWithX:0.0 Y:0.0 Z:0.0 W:0.0],
+                                 nil];
+
+        CIFilter *darkScratches = [CIFilter filterWithName:@"CIColorMatrix" keysAndValues:kCIInputImageKey, otherRandomImage,
+                                   @"inputRVector", [CIVector vectorWithX:3.659f Y:0.0 Z:0.0 W:0.0],
+                                   @"inputGVector", [CIVector vectorWithX:0.0 Y:0.0 Z:0.0 W:0.0],
+                                   @"inputBVector", [CIVector vectorWithX:0.0 Y:0.0 Z:0.0 W:0.0],
+                                   @"inputAVector", [CIVector vectorWithX:0.0 Y:0.0 Z:0.0 W:0.0],
+                                   @"inputBiasVector", [CIVector vectorWithX:0.0 Y:1.0 Z:1.0 W:1.0],
+                                   nil];
+
+        CIFilter *minimumComponent = [CIFilter filterWithName:@"CIMinimumComponent"];
+
+        CIFilter *composite = [CIFilter filterWithName:@"CIMultiplyCompositing"];
+
+        if (sepiaFilter && randomFilter && whiteSpecks && darkScratches && minimumComponent && composite) {
+            CIImage *sepiaImage = sepiaFilter.outputImage;
+            CIImage *whiteSpecksImage = [whiteSpecks.outputImage imageByCroppingToRect:sourceCIImage.extent];
+
+            CIImage *sepiaPlusWhiteSpecksImage = [CIFilter filterWithName:@"CISourceOverCompositing" keysAndValues:
+                                                  kCIInputImageKey, whiteSpecksImage,
+                                                  kCIInputBackgroundImageKey, sepiaImage,
+                                                  nil].outputImage;
+
+            CIImage *darkScratchesImage = [darkScratches.outputImage imageByCroppingToRect:sourceCIImage.extent];
+
+            [minimumComponent setValue:darkScratchesImage forKey:kCIInputImageKey];
+            darkScratchesImage = minimumComponent.outputImage;
+
+            [composite setValue:sepiaPlusWhiteSpecksImage forKey:kCIInputImageKey];
+            [composite setValue:darkScratchesImage forKey:kCIInputBackgroundImageKey];
+
+            [self addCIImageToCollectionView:composite.outputImage withFilterTitle:NSLocalizedString(@"Film", @"Film Filter")];
+        }
+    }];
 }
```

We start off by creating a few filters and images. The **`CISepiaTone`** filter will make a sepia tone version of our source image.

We then use **`CIRandomGenerator`** to make a random image called **`randomImage`**. Its output looks like color TV static:

We take this image and stretch it a little bit horizontally, and a *lot* vertically. This second image is called **`otherRandomImage`**.

We then create two filters using **`CIColorMatrix`**. **`whiteSpecks`** will pull out white specks from **`randomImage`**, and **`darkScratches`** will pull out vertical scratches from **`otherRandomImage`**.

We also create the filters **`minimumComponent`** and **`composite`** which we'll need to combine the layers.

After checking to ensure that those filters all exist, we get started building the image. First, we apply the sepia filter. We then generate **`whiteSpecksImage`** (we need to crop it it the source image's size since the randomly generated image is infinite.) Then we create **`sepiaPlusWhiteSpecksImage`** by using **`CISourceOverCompositing`** to overlay the white specks on top of the sepia-toned image. We then create and add **`darkScratchesImage`** on top of the white specks, and call **`outputImage`** on the composite filter to get the whole thing.

If you feel overwhelmed, that's OK - there are whole books written on Core Image. If you're interested in more information, check out **Core Image Programming Guide: Processing Images**.

Run the app again - you should now be able to see all the different filters, and apply them to an image.

Sending the Filtered Photo to Instagram

When the send button is pressed, let's check to make sure Instagram is installed, and ask the user if they'd like to write a caption:

BLCPostToInstagramViewController.m

```
+ - (void) sendButtonPressed:(id)sender {
+     NSURL *instagramURL = [NSURL URLWithString:@"instagram://location?id=1"];
+
+     if ([[UIApplication sharedApplication] canOpenURL:instagramURL]) {
+         UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"LOL" message:NSLocalizedString(@"Add a caption and send your imag
+         alert.alertViewStyle = UIAlertViewStylePlainTextInput;
+
+         UITextField *textField = [alert textFieldAtIndex:0];
+         textField.placeholder = NSLocalizedString(@"Caption", @"Caption");
+
+         [alert show];
+     } else {
+         UIAlertView *alert = [[UIAlertView alloc] initWithTitle:NSLocalizedString(@"No Instagram App", nil) message:NSLocalizedStri
+         [alert show];
+     }
+ }
```

On iOS, apps can define their own URL schemes so that other apps can open them. Instagram defines their own; so checking to see if the **`instagram://`** URL scheme can be handled is an easy way to tell if the app is installed.

Note that we set **`alert.alertViewStyle`** to **`UIAlertViewStylePlainTextInput`**. This will add a **`UITextField`** to the alert view.

To send the filtered photo to Instagram, we'll build a **`UIDocumentInteractionController`**. This is similar to the **`UIActivityViewController`** class you used earlier (for long-pressing on images in the feed view), but you pass in a file instead of a **`UIImage`**. We'll do this work after checking that the **OK** button was tapped:

BLCPostToInstagramViewController.m

```
+
+ - (void) alertView:(UIAlertView *)alertView didDismissWithButtonIndex:(NSInteger)buttonIndex {
+     if (buttonIndex != alertView.cancelButtonIndex) {
+
+         NSData *imagedata = UIImageJPEGRepresentation(self.previewImageView.image, 0.9f);
+
+         NSURL *tmpDirURL = [NSURL fileURLWithPath:NSTemporaryDirectory() isDirectory:YES];
+         NSURL *fileURL = [[tmpDirURL URLByAppendingPathComponent:@"blocstagram"] URLByAppendingPathExtension:@"igo"];
+
+         BOOL success = [imagedata writeToURL:fileURL atomically:YES];
+
+         if (!success) {
+             UIAlertView *alert = [[UIAlertView alloc] initWithTitle:NSLocalizedString(@"Couldn't save image", nil) message:NSLocali
+             [alert show];
+             return;
+         }
+
+         UIDocumentInteractionController *documentController = [UIDocumentInteractionController interactionControllerWithURL:fileURL
+         documentController.UTI = @"com.instagram.exclusivegram";
+
+         documentController.delegate = self;
+
+         NSString *caption = [alertView textFieldAtIndex:0].text;
+
+         if (caption.length > 0) {
+             documentController.annotation = @{@"InstagramCaption": caption};
+         }
+
+         if (self.sendButton.superview) {
+             [documentController presentOpenInMenuFromRect:self.sendButton.bounds inView:self.sendButton animated:YES];
+         } else {
+             [documentController presentOpenInMenuFromBarButtonItem:self.sendBarButton animated:YES];
+         }
+     }
+ }
```

This process is slightly more complex because we need to write the file to disk and send the URL of the file to Instagram. Essentially, we convert the image to **`NSData`**, create a file in the temporary directory with the **`igo`** ("Instagram only") extension and **`com.instagram.exclusivegram`** UTI, and initialize and present a **`UIDocumentInteractionController`** with that file.

When the document interaction controller finishes, we should dismiss the crop view:

BLCPostToInstagramViewController.m

```
+ #pragma mark - UIDocumentInteractionControllerDelegate
+
+ - (void)documentInteractionController:(UIDocumentInteractionController *)controller didEndSendingToApplication:(NSString *)applicat
+     [self dismissViewControllerAnimated:YES completion:nil];
+ }
```

Run the app on a device with Instagram installed. You can now send your photo to the Instagram app!

- Remove some of the layout code from this controller by replacing the vanilla **`UICollectionViewCell`** with a subclass. (Refer back to your **`UITableViewCell`** subclass for hints).
- Add two more filters to the collection view. Make at least one of them a compound filter.

Terminal

```
$ git add .
$ git commit -m 'Photo Filters and Sending to Instagram'
$ git checkout master
$ git merge filters-and-posting
$ git push
```

Submit this checkpoint's assignment with links for your repo and commit.

assignment completed

---

hello@bloc.io
Considering enrolling? (404) 480-2562
Partnership / Corporate Inquiries? (650) 741-5682

Roadmap · Previous Checkpoint Jump to Submission · Next Checkpoir

iPad Customization

Running the app on an iPad (or iPad Simulator) will result in a few issues:

1. The images in the feed are way too big, and take up almost the whole screen
2. The zoom logic for tapping on an image in the feed doesn't usually work well, since the screen resolution is higher than the image resolution
3. When tapping on the camera icon, presenting a full-screen view can feel clunky

To fix these, we'll modify some of our code to behave differently on iPad. We'll also update the camera button to use a popover on iPad.

cd into your Blocstagram directory and make a new **git** branch:

Terminal

```
$ git checkout -b ipad
```

Detecting the User's Device

Checking if the use is running your app on an iPhone is easy:

```
if ([[UIDevice currentDevice] userInterfaceIdiom] == UIUserInterfaceIdiomPhone) {
    // It's an iPhone
} else {
    // It's an iPad
}
```

This code is a little clunky, so we'll use **#define** to consolidate this code into one word.

Open up **Blocstagram-Prefix.pch**. Anything you put in this file will be compiled into all files in your project.

Blocstagram-Prefix.pch

```
#import <Availability.h>

+ #define isPhone ([[UIDevice currentDevice] userInterfaceIdiom] == UIUserInterfaceIdiomPhone)
+
#ifndef __IPHONE_3_0
#warning "This project uses features only available in iOS SDK 3.0 and later."
#endif
```

Now our above code will look like this:

```
if (isPhone) {
    // It's an iPhone
} else {
    // It's an iPad
}
```

Reasonable Sizing for Images

We'll need to update our auto layout constraints based on the device, but most of them will stay the same.

First, create separate methods for the image's horizontal constraint on each device:

BLCMediaTableViewCell.m

```
+ - (void) createConstraints {
+     if (isPhone) {
+         [self createPhoneConstraints];
+     } else {
+         [self createPadConstraints];
+     }
+
+     [self createCommonConstraints];
+ }
+
+ - (void) createPadConstraints {
+     NSDictionary *viewDictionary = NSDictionaryOfVariableBindings(_mediaImageView, _usernameAndCaptionLabel, _commentLabel, _likeBu
+
+     [self.contentView addConstraints:[NSLayoutConstraint constraintsWithVisualFormat:@"H:[_mediaImageView(==320)]" options:kNilOpti
+     [self.contentView addConstraint: [NSLayoutConstraint constraintWithItem:self.contentView
+                                                                   attribute:NSLayoutAttributeCenterX
+                                                                   relatedBy:0
+                                                                      toItem:_mediaImageView
+                                                                   attribute:NSLayoutAttributeCenterX
+                                                                  multiplier:1
+                                                                    constant:0]];
+
+ }
+
+ - (void) createPhoneConstraints {
+     NSDictionary *viewDictionary = NSDictionaryOfVariableBindings(_mediaImageView, _usernameAndCaptionLabel, _commentLabel, _likeBu
+
+     [self.contentView addConstraints:[NSLayoutConstraint constraintsWithVisualFormat:@"H:|[_mediaImageView]|" options:kNilOptions m
+ }
```

On iPhone, the constraints will remain the same.

On iPad, the constraints will fix the image size at 320 points and center it on the screen.

Delete the horizontal image constraint from the initializer and call **`createConstraints`**:

BLCMediaTableViewCell.m

```
-         NSDictionary *viewDictionary = NSDictionaryOfVariableBindings(_mediaImageView, _usernameAndCaptionLabel, _commentLabel, _li
-
-         [self.contentView addConstraints:[NSLayoutConstraint constraintsWithVisualFormat:@"H:|[_mediaImageView]|" options:kNilOptio
+         [self createConstraints];
```

Delete the remaining constraints from the initializer:

BLCMediaTableViewCell.m

```
-         [self.contentView addConstraints:[NSLayoutConstraint constraintsWithVisualFormat:@"H:|[_commentLabel]|" options:kNilOptions
-         [self.contentView addConstraints:[NSLayoutConstraint constraintsWithVisualFormat:@"H:|[_commentView]|" options:kNilOptions
-
-         [self.contentView addConstraints:[NSLayoutConstraint constraintsWithVisualFormat:@"V:|[_mediaImageView][_usernameAndCaption
-                                                                                    options:kNilOptions
-                                                                                    metrics:nil
-                                                                                      views:viewDictionary]];
-
-         self.imageHeightConstraint = [NSLayoutConstraint constraintWithItem:_mediaImageView
-                                                                   attribute:NSLayoutAttributeHeight
-                                                                   relatedBy:NSLayoutRelationEqual
-                                                                      toItem:nil
-                                                                   attribute:NSLayoutAttributeNotAnAttribute
-                                                                  multiplier:1
-                                                                    constant:100];
-
-
-         self.usernameAndCaptionLabelHeightConstraint = [NSLayoutConstraint constraintWithItem:_usernameAndCaptionLabel
-                                                                                     attribute:NSLayoutAttributeHeight
-                                                                                     relatedBy:NSLayoutRelationEqual
-                                                                                        toItem:nil
-                                                                                     attribute:NSLayoutAttributeNotAnAttribute
-                                                                                    multiplier:1
-                                                                                      constant:100];
-
-         self.commentLabelHeightConstraint = [NSLayoutConstraint constraintWithItem:_commentLabel
-                                                                          attribute:NSLayoutAttributeHeight
-                                                                          relatedBy:NSLayoutRelationEqual
-                                                                             toItem:nil
-                                                                          attribute:NSLayoutAttributeNotAnAttribute
-                                                                         multiplier:1
-                                                                           constant:100];
-
-         [self.contentView addConstraints:@[self.imageHeightConstraint, self.usernameAndCaptionLabelHeightConstraint, self.commentLa
```

Add them back in **`createCommonConstraints`**:

BLCMediaTableViewCell.m

```
+     NSDictionary *viewDictionary = NSDictionaryOfVariableBindings(_mediaImageView, _usernameAndCaptionLabel, _commentLabel, _likeBu
+     [self.contentView addConstraints:[NSLayoutConstraint constraintsWithVisualFormat:@"H:|[_usernameAndCaptionLabel][_likeButton(==
+     [self.contentView addConstraints:[NSLayoutConstraint constraintsWithVisualFormat:@"H:|[_commentLabel]|" options:kNilOptions met
+     [self.contentView addConstraints:[NSLayoutConstraint constraintsWithVisualFormat:@"H:|[_commentView]|" options:kNilOptions metr
+
+     [self.contentView addConstraints:[NSLayoutConstraint constraintsWithVisualFormat:@"V:|[_mediaImageView][_usernameAndCaptionLabe
+                                                                                options:kNilOptions
+                                                                                metrics:nil
+                                                                                  views:viewDictionary]];
+
+     self.imageHeightConstraint = [NSLayoutConstraint constraintWithItem:_mediaImageView
+                                                               attribute:NSLayoutAttributeHeight
+                                                               relatedBy:NSLayoutRelationEqual
+                                                                  toItem:nil
+                                                               attribute:NSLayoutAttributeNotAnAttribute
+                                                              multiplier:1
+                                                                constant:100];
+
+
+     self.usernameAndCaptionLabelHeightConstraint = [NSLayoutConstraint constraintWithItem:_usernameAndCaptionLabel
+                                                                                 attribute:NSLayoutAttributeHeight
+                                                                                 relatedBy:NSLayoutRelationEqual
+                                                                                    toItem:nil
+                                                                                 attribute:NSLayoutAttributeNotAnAttribute
+                                                                                multiplier:1
+                                                                                  constant:100];
+
+     self.commentLabelHeightConstraint = [NSLayoutConstraint constraintWithItem:_commentLabel
+                                                                      attribute:NSLayoutAttributeHeight
+                                                                      relatedBy:NSLayoutRelationEqual
+                                                                         toItem:nil
+                                                                      attribute:NSLayoutAttributeNotAnAttribute
+                                                                     multiplier:1
+                                                                       constant:100];
+
+     [self.contentView addConstraints:@[self.imageHeightConstraint, self.usernameAndCaptionLabelHeightConstraint, self.commentLabelH
+ }
```

That will take care of adjusting the width and positioning on each device. We just need to take care of the height now, which we can easily do in **`layoutSubviews`**:

BLCMediaTableViewCell.m

```
     if (_mediaItem.image) {
-        self.imageHeightConstraint.constant = self.mediaItem.image.size.height / self.mediaItem.image.size.width * CGRectGetWidth(s
+        if (isPhone) {
+            self.imageHeightConstraint.constant = self.mediaItem.image.size.height / self.mediaItem.image.size.width * CGRectGetWid
+        } else {
+            self.imageHeightConstraint.constant = 320;
+        }
     } else {
         self.imageHeightConstraint.constant = 0;
     }
```

Run the app. The images will now be reasonably sized, and centered in the middle of the view.

Presenting Images in a Form Sheet View

Since the images won't be as high-resolution as the screen, we don't want to display them full-screen. A good option is to present them using a "form sheet" view, which creates a medium-sized rectangle in the middle of the screen.

All we need to do is change the presentation style. Open up **`[BLCImagesTableViewController -cell:didTapImageView:]`**:

BLCImagesTableViewController.m

```
-      fullScreenVC.transitioningDelegate = self;
-      fullScreenVC.modalPresentationStyle = UIModalPresentationCustom;
+      if (isPhone) {
+          fullScreenVC.transitioningDelegate = self;
+          fullScreenVC.modalPresentationStyle = UIModalPresentationCustom;
+      } else {
+          fullScreenVC.modalPresentationStyle = UIModalPresentationFormSheet;
+      }

      [self presentViewController:fullScreenVC animated:YES completion:nil];
```

Run the app and tap an image. The image will now appear in the middle of the screen, and can still be panned and zoomed as normal.

It would be nice if the view would dismiss when the user taps on the gray border around the image, so let's add that while we're here.

To accomplish this, we'll add a tap gesture recognizer to the app's window. It will only fire if the user taps on an area with no other touch ever handling (namely, outside the view controller's view).

Add a property for the gesture recognizer:

BLCMediaFullScreenViewController.m

```
@property (nonatomic, strong) UITapGestureRecognizer *tap;
@property (nonatomic, strong) UITapGestureRecognizer *doubleTap;

+ @property (nonatomic, strong) UITapGestureRecognizer *tapBehind;
+
@end
```

In **viewDidLoad**, conditionally instantiate the gesture recognizer:

BLCMediaFullScreenViewController.m

```
      [self.tap requireGestureRecognizerToFail:self.doubleTap];

+      if (isPhone == NO) {
+          self.tapBehind = [[UITapGestureRecognizer alloc] initWithTarget:self action:@selector(tapBehindFired:)];
+          self.tapBehind.cancelsTouchesInView = NO;
+      }
+
      [self.scrollView addGestureRecognizer:self.tap];
      [self.scrollView addGestureRecognizer:self.doubleTap];
```

In **viewWillAppear:**, add it to the window:

BLCMediaFullScreenViewController.m

```
      [super viewWillAppear:animated];

      [self centerScrollView];
+
+      if (isPhone == NO) {
+          [[[[UIApplication sharedApplication] delegate] window] addGestureRecognizer:self.tapBehind];
+      }
  }
```

In **viewWillDisappear:**, remove it from the window.

BLCMediaFullScreenViewController.m

```
+     [super viewWillDisappear:animated];
+
+     if (isPhone == NO) {
+         [[[[UIApplication sharedApplication] delegate] window] removeGestureRecognizer:self.tapBehind];
+     }
+ }
```

Finally, implement **`tapBehindFired:`** to dismiss the controller:

BLCMediaFullScreenViewController.m

```
+ - (void) tapBehindFired:(UITapGestureRecognizer *)sender {
+     if (sender.state == UIGestureRecognizerStateEnded) {
+         CGPoint location = [sender locationInView:nil]; // Passing nil gives us coordinates in the window
+         CGPoint locationInVC = [self.presentedViewController.view convertPoint:location fromView:self.view.window];
+
+         if ([self.presentedViewController.view pointInside:locationInVC withEvent:nil] == NO) {
+             // The tap was outside the VC's view
+
+             if (self.presentingViewController) {
+                 [self dismissViewControllerAnimated:YES completion:nil];
+             }
+         }
+     }
+ }
```

Run the app again. You should now be able to tap on the gray area outside of the image controller in order to dismiss it.

Presenting the Camera and Image Library in a Popover

A popover controller is a common tool used in iPad apps for displaying information without hiding the primary view from the user. It takes advantage of iPad's plentiful screen space to provide additional context to the user.

We'll use an **`NSNotification`** to dismiss the popover when the user's done.

Before we create the popover, let's set up the notification:

BLCDataSource.h

```
 @interface BLCDataSource : NSObject

+ extern NSString *const BLCImageFinishedNotification;
+
 + (instancetype) sharedInstance;
 + (NSString *) instagramClientID;
```

BLCDataSource.m

```
 @implementation BLCDataSource

+ NSString *const BLCImageFinishedNotification = @"BLCImageFinishedNotification";
+
 + (instancetype) sharedInstance {
```

When the image sharing has completed, we'll post a notification. Import **`BLCDataSource`** to get the notification name:

BLCPostToInstagramViewController.m

```
 #import "BLCPostToInstagramViewController.h"
+ #import "BLCDataSource.h"
```

BLCDataSource.m

```
- (void)documentInteractionController:(UIDocumentInteractionController *)controller didEndSendingToApplication:(NSString *)applicatio
-     [self dismissViewControllerAnimated:YES completion:nil];
+     [[NSNotificationCenter defaultCenter] postNotificationName:BLCImageFinishedNotification object:self];
 }
```

Now, let's create the popover in **`BLCImagesTableViewController.m`**. Start by creating a property for the popover controller:

BLCImagesTableViewController.m

```
@property (nonatomic, assign) CGFloat lastKeyboardAdjustment;

+ @property (nonatomic, strong) UIPopoverController *cameraPopover;
+
@end
```

At the end of **`viewDidLoad`**, register this class as an observer for the notification:

BLCImagesTableViewController.m

```
                                          selector:@selector(keyboardWillHide:)
                                              name:UIKeyboardWillHideNotification
                                            object:nil];
+
+     [[NSNotificationCenter defaultCenter] addObserver:self
+                                              selector:@selector(imageDidFinish:)
+                                                  name:BLCImageFinishedNotification
+                                                object:nil];
```

In **`cameraPressed:`**, we'll continue to present the view controller as normal on iPhone, and we'll make a popover on iPad:

BLCImagesTableViewController.m

```
    if (imageVC) {
        UINavigationController *nav = [[UINavigationController alloc] initWithRootViewController:imageVC];
-         [self presentViewController:nav animated:YES completion:nil];
+
+         if (isPhone) {
+             [self presentViewController:nav animated:YES completion:nil];
+         } else {
+             self.cameraPopover = [[UIPopoverController alloc] initWithContentViewController:nav];
+             self.cameraPopover.popoverContentSize = CGSizeMake(320, 568);
+             [self.cameraPopover presentPopoverFromBarButtonItem:sender permittedArrowDirections:UIPopoverArrowDirectionAny animated
+         }
    }
```

When initializing a popover controller, you must always provide the content view controller, which we're doing in **`initWithContentViewController:`**. You can change this later by assigning a new view controller to the popover controller's **`contentViewController`** property.

After initializing the popover controller, we present it from the camera button. Popover controllers can also be presented from a specific **`CGRect`** within a view. When the iPad is rotated, popovers presented with bar button items will remain anchored to the button. However, popovers presented from a rect require manual positioning during rotation, so whenever possible you should use bar button items.

If one of our controllers dismisses without an image, we'll need to dismiss the popover:

BLCImagesTableViewController.m

```
    } else {
-         [nav dismissViewControllerAnimated:YES completion:nil];
+         if (isPhone) {
+             [nav dismissViewControllerAnimated:YES completion:nil];
+         } else {
+             [self.cameraPopover dismissPopoverAnimated:YES];
+             self.cameraPopover = nil;
+         }
     }
 }
```

Finally, we'll need to dismiss the popover once we've sent the image to Instagram:

BLCImagesTableViewController.m

```
+ #pragma mark - Popover Handling
+
+ - (void) imageDidFinish:(NSNotification *)notification {
+     if (isPhone) {
+         [self dismissViewControllerAnimated:YES completion:nil];
+     } else {
+         [self.cameraPopover dismissPopoverAnimated:YES];
+         self.cameraPopover = nil;
+     }
+ }
```

We have one final change to make. On iPad, **`UIDocumentInteractionController`** presents as a popover. If a popover is deallocated while it's still visible, the app will crash with an error like this:

> *Terminating app due to uncaught exception 'NSGenericException', reason: '-[UIPopoverController dealloc] reached while popover is still visible.'*

To fix this, we'll create a property for the document interaction controller. Open up **`BLCPostToInstagramViewController.m`** and make a property for it:

BLCPostToInstagramViewController.m

```
@property (nonatomic, strong) UIButton *sendButton;
@property (nonatomic, strong) UIBarButtonItem *sendBarButton;

+ @property (nonatomic, strong) UIDocumentInteractionController *documentController;
+
@end
```

Change **`documentController`** references to **`self.documentController`**:

BLCPostToInstagramViewController.m

```
-           documentController.UTI = @"com.instagram.exclusivegram";
+           self.documentController = [UIDocumentInteractionController interactionControllerWithURL:fileURL];
+           self.documentController.UTI = @"com.instagram.exclusivegram";

-           documentController.delegate = self;
+           self.documentController.delegate = self;

        NSString *caption = [alertView textFieldAtIndex:0].text;

        if (caption.length > 0) {
-             documentController.annotation = @{@"InstagramCaption": caption};
+             self.documentController.annotation = @{@"InstagramCaption": caption};
        }

        if (self.sendButton.superview) {
-             [documentController presentOpenInMenuFromRect:self.sendButton.bounds inView:self.sendButton animated:YES];
+             [self.documentController presentOpenInMenuFromRect:self.sendButton.bounds inView:self.sendButton animated:YES];
        } else {
-             [documentController presentOpenInMenuFromBarButtonItem:self.sendBarButton animated:YES];
+             [self.documentController presentOpenInMenuFromBarButtonItem:self.sendBarButton animated:YES];
```

Run the app on your iPad or iPad Simulator. You should now be able to capture, choose, crop, filter, and send photos to Instagram all within the popover.

In `[BLCImagesTableViewController -cell: didLongPressImageView:]`, a `UIActivityViewController` is presented on the screen. Update the code to present this controller in a popover instead. Ensure the popover is placed correctly after a rotation.

You may need to implement the `UIViewController` method `willAnimateRotationToInterfaceOrientation:duration:` to re-present or dismiss the popover because the image will have moved and may or may not be on screen.

When you're satisfied with your project and the assignment, commit, merge and push your code to GitHub:

Terminal

```
$ git add .
$ git commit -m 'iPad Customization'
$ git checkout master
$ git merge ipad
$ git push
```

Submit this checkpoint's assignment with links for your repo and commit.

assignment completed

COURSES

- Full Stack Web Development
- Frontend Web Development
- UX Design
- Android Development
- iOS Development

ABOUT BLOC

- Our Team | Jobs
- Bloc Veterans Program
- Employer Sponsored
- FAQ
- Blog
- Engineering Blog
- Refer-a-Friend
- Privacy Policy
- Terms of Service

MADE BY BLOC

- Tech Talks & Resources
- Programming Bootcamp Comparison
- Jottly: A Beginner's Guide to HTML, CSS, Skeleton and Animate.css
- Swiftris: Build Your First iOS Game with Swift
- Webflow Tutorial: Design Responsive Sites with Webflow
- Ruby Warrior
- Bloc's Diversity Scholarship

SIGN UP FOR OUR MAILING LIST

Send

hello@bloc.io

Considering enrolling? (404) 480-2562

Partnership / Corporate Inquiries? (650) 741-5682

Writing Unit Tests

Unit tests are small methods written to automatically test that another method works. They are called **unit** tests because they test a **unit** of your code - that is, a small chunk of it.

This checkpoint won't venture too far into the world of automated testing, but there are some helpful resources listed at the end if you'd like to read more.

Unit Tests: What Are They Good For?

Unit tests are primarily used to ensure the long-term stability of your app. Consider the following scenario:

1. You add a feature to allow users to look at Instagram user profiles
2. Later, you add a feature to allow users to follow or unfollow users

You'll want to ensure that introducing feature #2 doesn't break feature #1. If feature #1 is broken by feature #2, this is called a **regression**. The amount of code you have covered by automated tests is called your **test coverage**.

The purpose of automated tests is to prevent regressions. Unit testing is one type of automated testing.

> ### *Other Types of Automated Testing*
>
> - ***Automated Code Auditing*** *tests the style ar[?]eup of your code. For example, tests could be written to ensure classes don't get too big, or that naming convention[?]llowed.*
> - ***Automated Functional Testing*** *actually runs your app on a device or simulator and steps through features as a user would do. For example, a test could be written to ensure that the user is able to login successfully. (Online services like* ***AppThwack*** *allow you to test your app on hundreds of real devices online.)*
> - ***Automated Stress Testing*** *tests your app in different high-stress scenarios, such as very rapid user input or very constrained Internet bandwidth.*

Tests don't add features to your app, but they can make you more comfortable adding features quickly without doing tons of grueling manua testing.

Let's write some. **`cd`** into your Blocstagram directory and make a new **`git`** branch:

Terminal

```
$ git checkout -b unit-tests
```

Writing Unit Tests

Xcode has a built-in testing framework called **`XCTest`**. Let's start by creating a unit test for **`BLCUser`** objects.

1. In the Project Navigator, select the **`BlocstagramTests`** folder.
2. Press **File** > **New** > **File...** or ⌘**N**.

5. Make sure that only the **BlocstagramTests** target is checked, and press **Create**.

> *Checking only the* **BlocstagramTests** *target ensures that the unit tests aren't compiled into your app when you ship it.*

BLCUserTests.m should be open. As you'll remember from the early exercises checkpoints, the methods run as follows:

- **setUp** runs before each test
- **tearDown** runs after each test
- all methods beginning with **test** are considered tests and will run once each
- other methods are ignored unless you call them

Delete the automatically failing **testExample** method:

BLCUserTests.m

```
- - (void)testExample
- {
-     XCTFail(@"No implementation for \"%s\"", __PRETTY_FUNCTION__);
- }
```

Add a new method to test that the **BLCUser** initializer works as expected:

BLCUserTests.m

```
+ - (void)testThatInitializationWorks
+ {
+     NSDictionary *sourceDictionary = @{@"id": @"8675309",
+                                        @"username" : @"d'oh",
+                                        @"full_name" : @"Homer Simpson",
+                                        @"profile_picture" : @"http://www.example.com/example.jpg"};
+     BLCUser *testUser = [[BLCUser alloc] initWithDictionary:sourceDictionary];
+
+     XCTAssertEqualObjects(testUser.idNumber, sourceDictionary[@"id"], @"The ID number should be equal");
+     XCTAssertEqualObjects(testUser.userName, sourceDictionary[@"username"], @"The username should be equal");
+     XCTAssertEqualObjects(testUser.fullName, sourceDictionary[@"full_name"], @"The full name should be equal");
+     XCTAssertEqualObjects(testUser.profilePictureURL, [NSURL URLWithString:sourceDictionary[@"profile_picture"]], @"The profile pic
+ }
```

This method makes an **NSDictionary** that mimics the relevant portion of the response from the Instagram API. This dictionary is passed to **[BLCUser -initWithDictionary:]**. After the user is created, we then assert that the **testUser**'s four properties (**idNumber**, **userName**, **fullName**, and **profilePictureURL**) are what we expect them to be.

If any of these assertions are false, the test fails. If they're all true, the test passes.

Let's write another test for **BLCComment**. follow the same process, and use this test:

BLCCommentTests

```
+ - (void)testThatInitializationWorks
+ {
+     NSDictionary *sourceDictionary = @{@"id": @"8675309",
+                                        @"text" : @"Sample Comment"};
+
+     BLCComment *testComment = [[BLCComment alloc] initWithDictionary:sourceDictionary];
+
+     XCTAssertEqualObjects(testComment.idNumber, sourceDictionary[@"id"], @"The ID number should be equal");
+     XCTAssertEqualObjects(testComment.text, sourceDictionary[@"text"], @"The text should be equal");
+ }
```

In all of these tests we're using **XCTAssertEqualObjects**, which sends an **isEqual:** message to determine equality. There are many other assertions; here are some common ones:

- `XCTFail` generates a failure (typically used inside of an `if/else` block
- `XCTAssertNotNil` asserts that an object isn't `nil`.

A full list is **available here**.

When Should Unit Tests Be Written?

Most proponents of writing unit tests are also proponents of **Test-Driven Development** (or **TDD**).

When creating new classes, we've followed this pattern:

1. Write the public interface (in the `.h` file)
2. Write the private interface and implementation (in the `.m` file)

In TDD, you write the tests in between:

1. Write the public interface
2. Write the tests, all of which will fail
3. Write the implementation until all tests pass

To put it more simply: test-driven development is the process of writing failing tests before writing functional code, such that the functional code makes the tests pass. By following this process as you write code, you can substantially increase your test coverage. You can also gain a clearer idea of what the code you're about to write needs to to.

Additional Resources

If you're interested in learning more about automated testing, check out these articles:

- **Introduction to TDD**
- **Test-Driven Development on Wikipedia**; see especially the *Test-driven development cycle* section
- Apple's **About Testing with Xcode** manual
- The **Unit Test Your App** section of Apple's Xcode Overview manual
- objc.io's **Testing View Controllers** (it relies on other testing frameworks, but it's still interesting and helpful).

Roadmap · **Previous Checkpoint** **Jump to Submission** · **Next Checkpoir**

Your assignment | **Ask a question** | **Submit your work**

Write some additional unit tests:

1. Write a test for `BLCMedia`'s initializer.
2. Write a test that ensures `[BLCComposeCommentView -setText:]` sets `isWritingComment` to `YES` if there's text, and another to ensure that it's set to `NO` if there's not text.
3. Write a variety of tests to ensure that `[BLCMediaTableViewCell +heightForMediaItem:width:]` returns accurate heights. You may need to add sample images to your project's test bundle to accomplish this.

When you're satisfied with your project and the assignment, commit, merge and push your code to GitHub:

Terminal

```
$ git add .
$ git commit -m 'Added Unit Tests'
$ git checkout master
$ git merge unit-tests
$ git push
```

Submit this checkpoint's assignment with links for your repo and commit.

assignment completed

Roadmap · Previous Checkpoint Jump to Submission · Next Checkpoir

Reducing Bugs and Improving User Experience

Although we wish it weren't true, **all non-trivial software has bugs**. In addition to manual and automated testing, there are a few techniques you can use to minimize bugs - or when they're present, to minimize their impact:

- **Static Analysis** runs all of your methods through a variety of checks for common bugs
- **Crash Reporters** send you a stack trace whenever your app crashes
- **Analytics Tools** let you see how people use your app

Just like the Unit Testing checkpoint, we'll give you some broad information and resources here, but this information isn't critical to building iOS apps so we won't spend a lot of time on it.

Static Analysis

What the heck is Static Analysis? **Wikipedia** calls it like it is:

> *Static program analysis is the analysis of computer software that is performed without actually executing programs. In most cases the analysis is performed on some version of the source code, and in the other cases, some form of the object code. The term is usually applied to the analysis performed by an automated tool.*

Xcode's built-in static analyzer, **Clang**, finds flaws in your code. This is especially helpful for subtle errors that slip by the compiler, so you won't see a normal yellow warning.

According to the **Xcode documentation**, common pro ? found include:

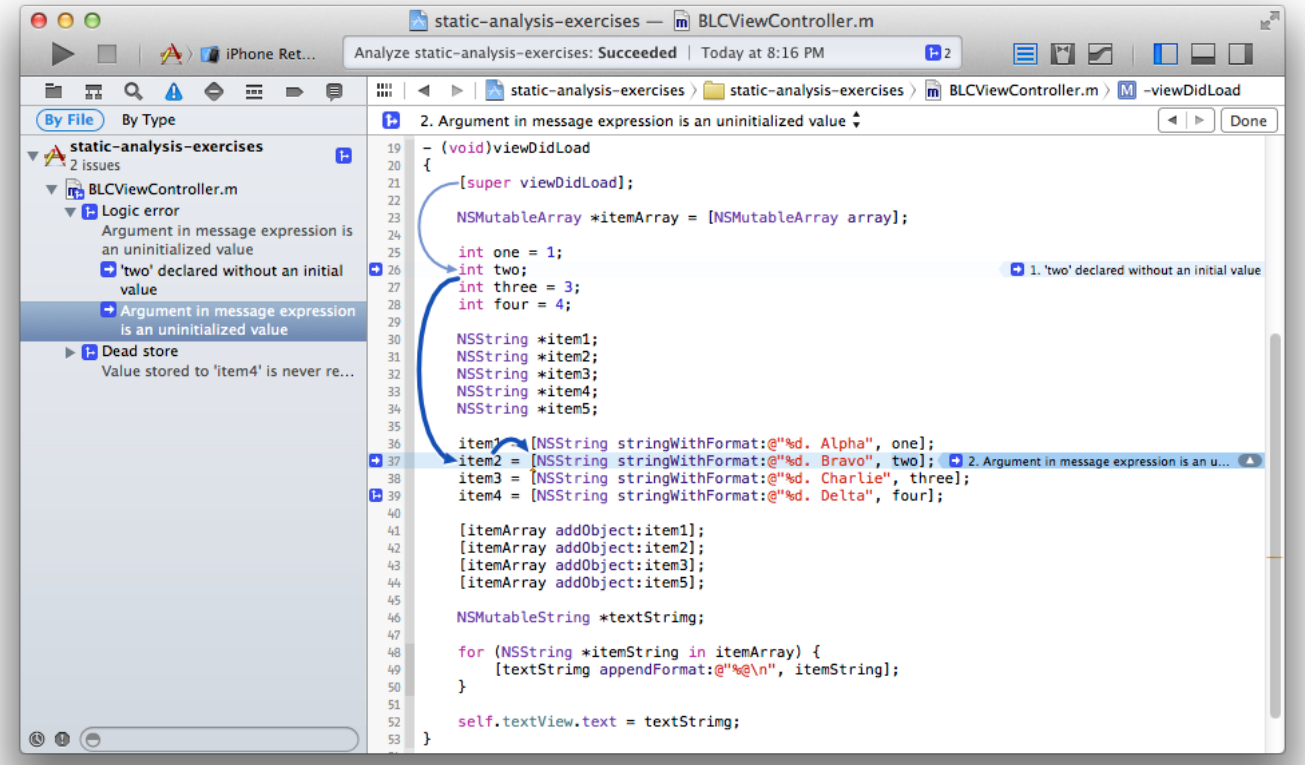
> - *Logic flaws, such as accessing uninitialized variables and dereferencing null pointers*
> - *Memory management flaws, such as leaking allocated memory*
> - *Dead store (unused variable) flaws*
> - *API-usage flaws that result from not following the policies required by the frameworks and libraries the project is using*

If you're feeling adventurous, check out this **list of Clang's checkers**. (Most of the interesting ones for you will be in the "OS X Checkers" section, which also applies to iOS.)

To run the analysis, press **Product** > **Analyze**, or ⇧⌘**B**. Go ahead and run it in your Blocstagram project. If it finds any errors, try to fix them. I not, you'll get a chance to fix some in the assignment.

If it finds no results, that's great!

Here's how the static analysis tool displays errors, from an example project which does have some issues:

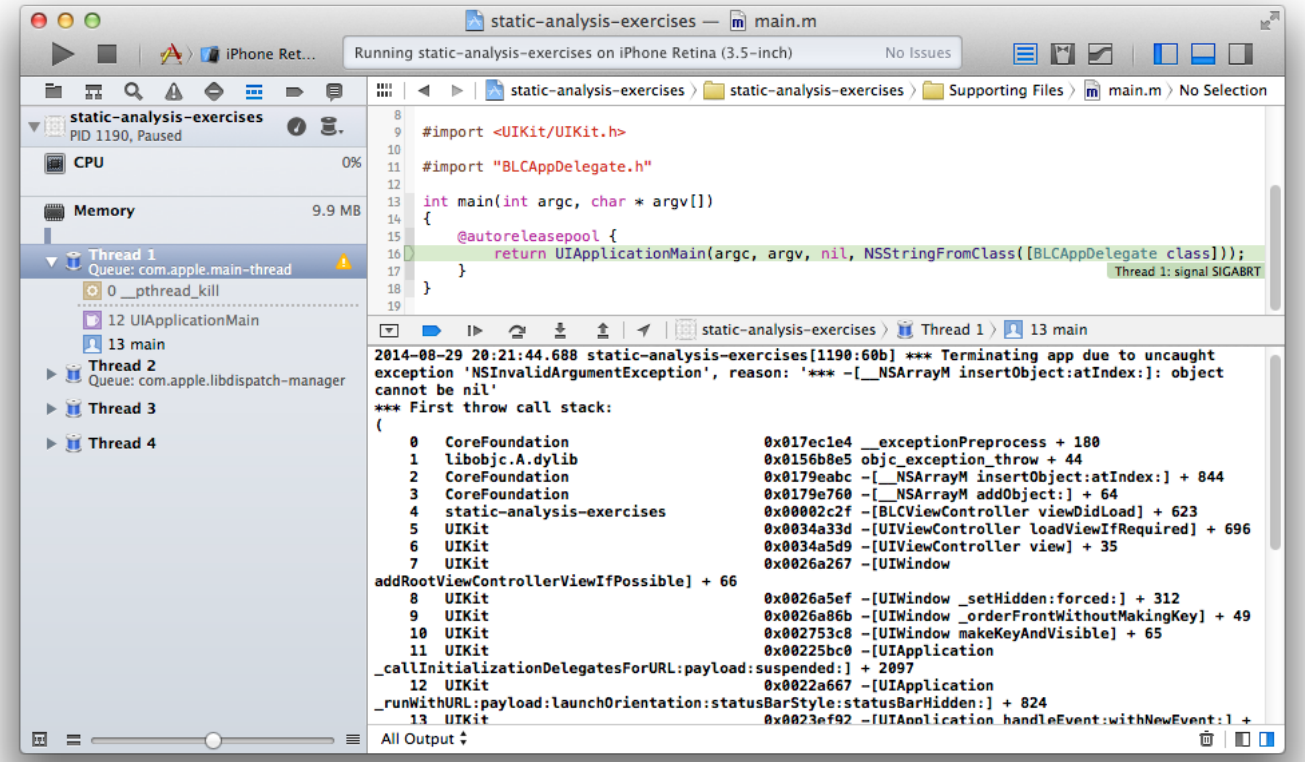


In this example, the **`int`** called **`two`** is not initialized when it's declared. It's then read from later, which will result in a garbage value.

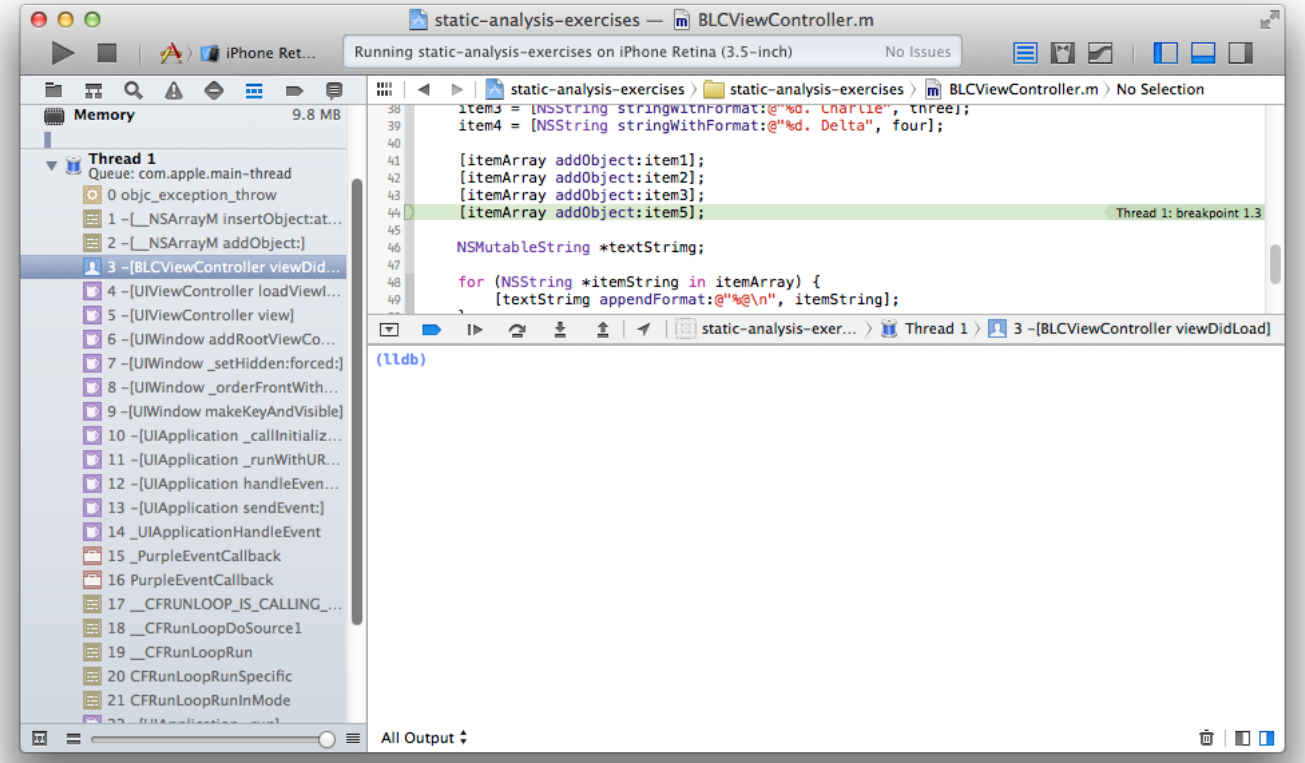
Note: Certain problems prevent Clang from seeing other problems. After resolving some errors, always run Clang again to see if there are more.

Crash Reporters

When your app crashes during development, you can see a stack trace printed to the Xcode console. Here's a crash for attempting to insert a **`nil`** object in a mutable array:



If you **add an exception breakpoint**, you can even find the exact line of your code that causes the crash:

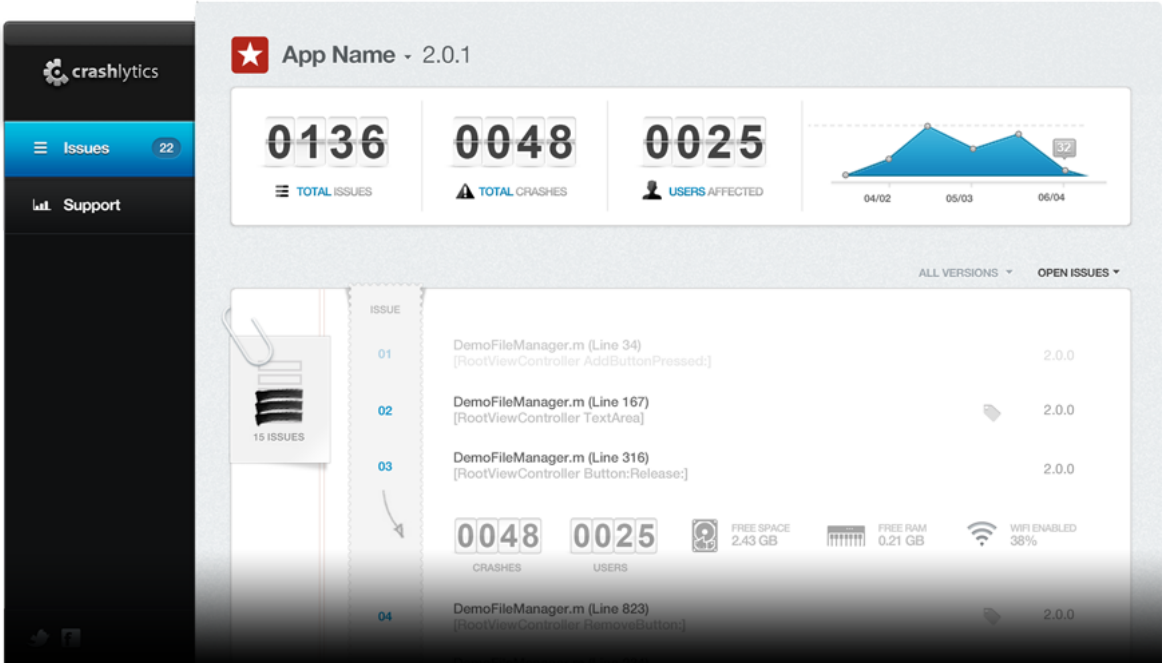


Alas, you can't see this stuff when your app is deployed to the App Store and crashes on an actual user's device.

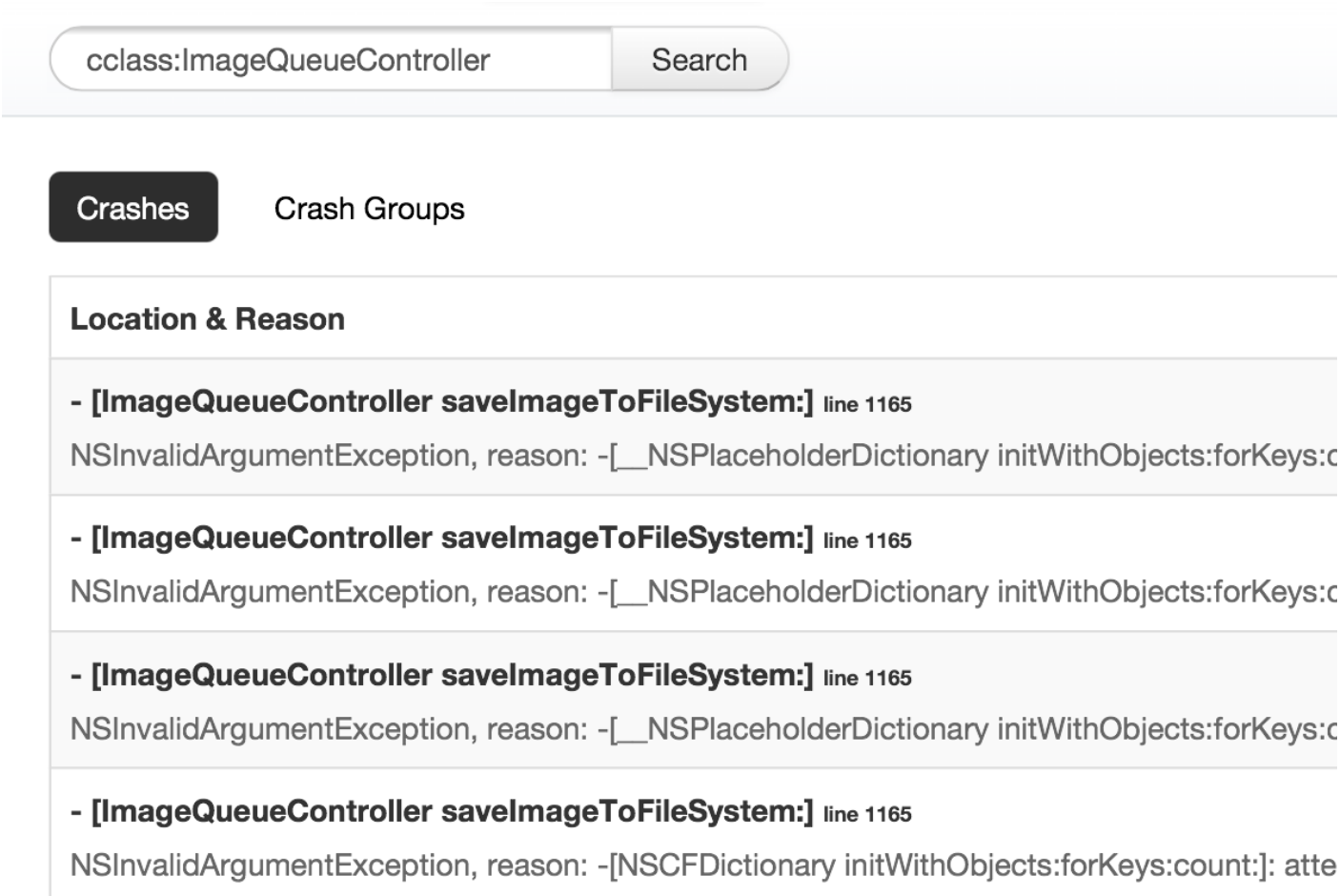
1. A reporting library that collects crash data when your app crashes
2. A web site that you can log into to view crashes

There are a number of crash reporting tools out there. The two most common are **Crashlytics** and **HockeyApp**.

Here's a report Crashlytics makes that helps you judge the severity of your crashes. The most common crashes appear at the top:



HockeyApp also provides a similar tool:



crash reporter per app, so choose carefully.

Analytics Tools

Crash reporters only tell you when something went wrong. Analytics tools allow you to see what your users are doing. There is some overlap between analytics tools and crash reporting tools; for example, HockeyApp provides both.

The two most common analytics tools are **Flurry Analytics** and **Google Analytics**.

Like crash reporters, all of these analytics companies provide excellent tutorials for their products, so we won't go into great detail here. We will, however, review the main features and use cases.

We'll show Flurry's syntax here, but most other analytics providers use similar syntax and terminology.

Events

When a user does something you want to track in your app, you log an *event*. This can be triggered by anything you can represent in code: a button tap, a login, a pinch, a network request, etc. For example, if you wanted to log how often users are sending photos to the Instagram app using Flurry, you could write:

```
[Flurry logEvent:@"Sent Photo to Instagram App"];
```

If you want to track some additional information, you can include parameters. For example, if you wanted to see which filter is the most popular:

```
NSDictionary *params = @{@"Filter" : /* filter name, like Noir or Drunk */};
[Flurry logEvent:@"Sent Photo to Instagram App" withParameters:params];
```

Users

You can generate aggregate information about your user base (if you collect this info from your users). You can also set a user ID. For example:

```
[Flurry setAge:21];
[Flurry setGender:@"m"];
[Flurry setUserID:@"USER_ID"];
```

This could allow you to determine if certain filters or more popular for different age groups, genders, locations, etc.

Errors

When an error occurs, your app should usually inform the user or attempt to retry or recover in some way. You can also log the error; this car let you know if there are errors occurring which you didn't account for, which a crash reporter wouldn't catch:

```
NSString *fullPath = /* a file path */
NSData *mediaItemData = /* some NSData */
NSError *dataError;

BOOL wroteSuccessfully = [mediaItemData writeToFile:fullPath options:NSDataWritingAtomic | NSDataWritingFileProtectionCompleteUnlessOp
if (!wroteSuccessfully) {
    [FlurryAnalytics logError:@"File didn't save" message:error.localizedDescription error:error];
}
```

By periodically checking which errors occur, you can ensure your code appropriately handles all possible cases.

Funnels

Funnels are helpful for tracking a series of related events. Flurry **describes them**:

> *Funnel analyses track users as they execute a defined set of steps, allowing you to see how many of the users that started a given process completed it.*

Configuring an event funnel for the photo posting flow in Blocstagram could allow you to answer questions like:

- How many users who tap the camera button eventually post an image?
- How many users are using an iOS device with no camera?
- How many users post pictures from the camera vs. their image library?

Knowing answers to questions like these can help inform your future development efforts.

Privacy Concerns

When sending data about your users, it's important to exercise a great amount of care and respect for the user's privacy. Here are some best practices:

- Never send authentication information like passwords or access tokens to an analytics or crash reporting service. For example, if you're sending a URL, make sure to strip out the access token. This information should only ever be stored in the user's keychain.
- If you send personally identifiable information, ensure that your users consent to this and are able to opt out.
- Your app should have a clear privacy policy that explains what information you collect and how you use it.

1. Create a new fork of the **`ios-static-analysis-exercises`** project. (Review the Git checkpoint if you need a refresher on the steps.) Run Build and Analyze to find issues with the code in **`BLCViewController.m`**. Fix the issues, and repeat until no more issues are found. When you're satisfied with your assignment, commit, merge and push your code to GitHub. Submit this checkpoint's assignment with links for your repo and commit.
2. Research some of the crash reporting and analytics tools available. In 250-500 words, write a few paragraphs comparing them and describing how you might use them in an app. Send this report as a message to your mentor.

assignment completed

UX Design
Android Development
iOS Development

ABOUT BLOC

Our Team | Jobs
Bloc Veterans Program
Employer Sponsored
FAQ
Blog
Engineering Blog
Refer-a-Friend
Privacy Policy
Terms of Service

MADE BY BLOC

Tech Talks & Resources
Programming Bootcamp Comparison
Jottly: A Beginner's Guide to HTML, CSS, Skeleton and Animate.css
Swiftris: Build Your First iOS Game with Swift
Webflow Tutorial: Design Responsive Sites with Webflow
Ruby Warrior
Bloc's Diversity Scholarship

SIGN UP FOR OUR MAILING LIST

Send

hello@bloc.io
Considering enrolling? (404) 480-2562
Partnership / Corporate Inquiries? (650) 741-5682

Roadmap · Previous Checkpoint Jump to Submission · Next Checkpoir

Preparing and Submitting your App to the App Store

You've written your app, you've tested it, you've added unit tests, and fixed all the static analysis issues. It's time to prep your app for submission to the iOS App Store!

Before you go any further, read these two documents:

- **App Store Review Guidelines** - it's long, but read the whole thing
- **Common App Rejections**

In its current state, Blocstagram might not get approved for these reasons:

> - *If ... you're trying to get your first practice App into the store to impress your friends, please brace yourself for rejection.*
> - *Apps that are not ... unique ... may be rejected*

If you want, you'll be able to continue working on Blocstagram in the projects section.

In this checkpoint, we'll select icons and launch images for Blocstagram, and if you're enrolled in the iOS Developer Program, we'll show you how to submit your app to the App Store.

Launch Images

When the user taps on your app, it launches immediately. However, it may take a few seconds for launch to complete. Remember all the stuf that needs to get done:

- Whatever work you have in `application:didFinishLaunchingWithOptions:`
- Building your view controller hierarchy
- Configuring, displaying, and laying out your views

During this time, in order to make your app look faster than it actually is, iOS displays your *launch image*. This image should look like the initia view the user sees, but without any content.

For example, here's the launch image for iOS 7's Weather app:

You must create launch images for all supported devices. If your app supports iPad, you also need to provide launch images for both landscape and portrait orientation.

The launch images documentation provides a list of all required resolutions. It also has a list of things *not* to include in your launch images **Don't** include:

> - *An "app entry experience," such as a splash screen*
> - *An About window*
> - *Branding elements, unless they are a static part of your app's first screen*

To add launch images, open your **`Images.xcassets`** file, select **`LaunchImage`**, and drag your image to the appropriate box. **See step-by-step instructions here**.

One technique for creating launch images is to hide all of your content and take a screen shot. To do this, set your views' **`hidden`** properties to **`YES`**, run your app in the appropriate Simulator, and press **File** > **Save Screen Shot**. Others prefer to create their launch images in graphic design applications.

App Icons

Similar to launch images, you must provide a variety of icons for your app. In addition to the home screen, **iOS uses your icon in other places**:

> *iOS can use versions of the app icon in Game Center, search results, Settings, and to represent app-created documents.*

blurry lines or uneven spacing caused by antialiasing. If possible, include additional detail on the larger versions.

The documentation lists all of the possible sizes. Add icons to `Images.xcassets`'s `AppIcon` Image Set, which will show you all of the appropriate sizes for your app.

In addition to the icon set, add a 1024x1024 icon called `iTunesArtwork@2x` and a 512x512 version called `iTunesArtwork` to your app.

Screenshots

You will need to have at least one screenshot for each supported device type available when you create your app in iTunes Connect. You can have up to 5 screenshots for each device type.

The three device types are:

- 3.5-Inch Retina Display Screenshots
- 4-Inch Retina Display Screenshots
- iPad Screenshots

You'll upload them in a bit.

The First App Submission

Now that you've added your icons and launch images, you're ready to submit your app to the App Store. The submission process is mostly done through **iTunes Connect**, a website for managing your content that's available in the iTunes store. You'll also need to use the **iOS Dev Center** and Xcode.

If you have an iOS Developer Account, feel free to follow along in your account.

There are a few steps you'll need to take the first time you submit an app. Let's take care of those first.

> ***Warning:*** *This process is widely considered to be incredibly annoying. You may want to step through it with your mentor.*

Creating an iTunes Connect record

The first time you submit any app, you need to **Create an iTunes Connect Record** for it:

1. Login to **iTunes Connect**
2. Click **My Apps**.
3. Click **Add New App**.
4. Select a default language, app name (as it should appear in the App Store), and any SKU number.
5. Select the **Bundle Identifier** from your project settings, like `com.Bloc.Blocstagram`. (If you haven't registered it yet, follow the link to register it; it may take a few minutes for this to propagate to iTunes Connect).
6. Press **Continue**.
7. iTunes Connect will ask you for a variety of information, including availability date, pricing, copyright information, and other information that affects how it appears in the iOS App Store. **This helpful table** shows how each option corresponds to the App Store listing.
8. At the end you'll be able to upload your screenshots and large app icon. (You can also skip most of these, but you'll need to upload at least one large app icon and one screenshot before submitting your app.)
9. Press **Save**.

You now have an app record in iTunes Connect!

Creating a Signing Identity and Provisioning Profile

1. Login to the **iOS Dev Center**.
2. Select **Certificates, Identifiers & Profiles**.

5. Select your **App ID** and press **Continue**.
6. Select your **iOS Distribution Certificate** and press **Continue**.
7. Type a profile name, like "My Awesome Profile" and press **Generate**.

Loading your Provisioning Profile in Xcode

Open Xcode and press **Xcode** > **Preferences...**.

Select **Accounts**, highlight your account, and press **View Details...**.

Press the reload icon until you see "My Awesome Profile" appear.

Press **Done** and close the preferences.

Steps You'll Complete Every Time

Every time you want to follow a new version, you'll roughly follow these steps. You can also edit your meta information and update version numbers in between builds.

1. Login to **iTunes Connect**
2. Click **My Apps** and select your app.
3. Select **View Details**, and press **Ready to Upload Binary**.
4. Answer the legal and compliance questions about this version of your app
5. Select **Continue**. Your app status should now be **Waiting For Upload**.

Now you'll need to create a signing identity.

Switch to Xcode. The rest of the submission process is completed in Xcode. It takes places in two parts: **validation** and **distribution**.

1. Make sure **iOS Device**, and not a simulator, is selected.
2. Press **Product** > **Archive**. (Archive is disabled if you skip step 1.)
3. The Organizer should open. If not, press **Window** > **Organizer**.
4. Select the build of your app and press **Validate...**.
5. Sign in and select "My Awesome Profile". Press **Validate**.

If there are any errors, you'll need to resolve them. Press **Finish**.

1. Select your build and press **Distribute.**
2. Select **Submit to the iOS App Store** and press **Next**.
3. Sign in, choose "My Awesome Profile" and press "Submit".

Your app will be uploaded and validated by Apple. You will usually hear back about whether your app was approved or not within a week; you can get a more accurate time estimate at **appreviewtimes.com**.

Read Apple's guidelines for **App Icons** and **Launch Images**. Send a message to your mentor with any questions.

assignment completed

COURSES

- Full Stack Web Development
- Frontend Web Development
- UX Design
- Android Development
- iOS Development

ABOUT BLOC

- Our Team | Jobs
- Bloc Veterans Program
- Employer Sponsored
- FAQ
- Blog
- Engineering Blog
- Refer-a-Friend
- Privacy Policy
- Terms of Service

MADE BY BLOC

- Tech Talks & Resources
- Programming Bootcamp Comparison
- Jottly: A Beginner's Guide to HTML, CSS, Skeleton and Animate.css
- Swiftris: Build Your First iOS Game with Swift
- Webflow Tutorial: Design Responsive Sites with Webflow
- Ruby Warrior
- Bloc's Diversity Scholarship

SIGN UP FOR OUR MAILING LIST

Send

- hello@bloc.io
- Considering enrolling? (404) 480-2562
- Partnership / Corporate Inquiries? (650) 741-5682

Roadmap · Previous Checkpoint Jump to Submission

Introduction to Projects

A large part of the growth you will experience at Bloc begins now. Projects focus on various aspects of iOS and programming which may have not been covered during the foundational phase. This is your chance to dig into the technologies that interest you and build apps from the ground up.

As you work through Bloc iOS projects, keep in mind the general workflow below and these best practices.

Best Practices

Keep your project in source control. Use the git flow, just like we've done in the apprenticeship so far. Keep finished code in your `master` branch. Keep separate features and bug fixes on separate branches until they're finished, then merge them into `master`.

Follow the Model/View/Controller pattern. Each class should be a model, a view, or a controller. Use patterns like delegation, notifications, and completion handlers to have these classes communicate. Consider creating a singleton data source controller (like `BLCDataSource` in Blocstagram) that mediates data between other controllers and the Internet (if your project uses the Internet).

Don't do too much at once. Remember **YAGNI** - You Aren't Gonna Need It. Use laser focus to work on one user story at a time, and write as little code as possible to get the job done without sacrificing quality.

General Workflow

There's no wrong order to do work in, but here is a com ? orkflow you can follow.

Initial Setup

- Read what the app does.
- Create a new Xcode project, git repository, and associated repository on GitHub.
- Create a `Podfile` with the required Cocoapods. For example, you'll probably need `AFNetworking`. Run `pod install` after the `Podfile` is created.
- Commit this initial code to your `master` branch.

Pick a User Story

- Pick **one** user story to work on. We'll use an authentication story as an example:

  > *As a user, I can enter my username and password to login.*

- Check out a feature branch with that name, like `login`:

- Break the user story into smaller tasks to make your work more manageable. This will help you think through your story in greater detai It's normal to rearrange tasks as you're working, but try to get the major pieces of work separated.

> *If you're having trouble thinking of subtasks, ask yourself these questions:*
>
> - *Does any work require the completion of other work? (Do you need username and password text boxes before you can test if they're correct?)*
> - *Are there any implied / unstated requirements? (Do you need Register or Forgot Password buttons?)*
> - *Will you need to write code that validates rules? (May a username be an e-mail address? Do passwords need to contain a number?)*

If you're working on a project in a group through **the Bloc Hacker Club**, you may want to discuss these questions with your team befor you begin coding.

Design the Data Model

Consider the **data model** and **communications** necessary for this story:

> - *How will we store the username and password before the user authenticates?*
> - *Once the user enters a username and password, how will we authorize this over the Internet?*
> - *If the login is successful, how will we store the authentication token?*
> - *How long should we store the authentication token?*

- Answer these questions first to yourself - by writing or thinking about them.

For example, answers might look like:

> - *How will we store the username and password before the user authenticates?*
>
> *The usernames and passwords do not require storage; they will be passed between appropriate methods until they are no longer needed.*
>
> - *Once the user enters a username and password, how will we authorize this over the Internet?*
>
> *We'll use* **`AFNetworking`** *to send a* **`POST`** *request to our API's authentication URL.*
>
> - *If the login is successful, how will we store the authentication token?*
>
> *While the app is running, the authentication token will be stored as a* **`property`** *on a data source singleton. The token will persist between launches because it is stored in the iOS Keychain. An* **`NSNotification`** *will inform interested objects that a token has become available.*
>
> - *How long should we store the authentication token?*
>
> *The token will be stored indefinitely unless an API response indicates that the token has expired. At that point, it will be removed from the iOS Keychain and the user will be asked to login again.*

Write Code to Match Your Design

Once you've written a plan, translate your answers into code. As literally as possible, consider this a translation from English into Objective-C or Swift.

For example, you now know:

- You'll need to create a method that submits the username and password to the API, and returns an auth token if it's successful. The method might have this signature in Objective-C:

```
- (void)loginWithEmail:(NSString *)email
              password:(NSString *)password
               success:(void (^)(NSString *token))success
               failure:(void (^)(NSError *error))failure;
```

  The function declaration could look like this in Swift:

```
func login(email:String, password:String, success: (token:String) -> Void, failure: (error:NSError?) -> Void) {

}
```

- You'll need code to save the token to the keychain.

The work for each project:

- Translate each section into a piece of code.
- Include automated test coverage for these methods.

Build a View and View Controller

Typically each story will require creating or modifying a view and view controller.

- Create a wireframe.
- Build the view controller and accompanying view to match your wireframe.

Remember the general flow for building view controllers.

- Create the views in Interface Builder, your Storyboard, or programmatically in **`viewDidLoad`**.
- Position your views in **`viewWillLayoutSubviews`** or add auto-layout constraints.
- Connect the appropriate views to the data methods created in the prior step.

> *For example, make a "Login" button that calls a method to verify method to verify the username and password entered.*

- Tweak the visual appearance - colors, padding, etc. - as appropriate.

Test Your Code

- See if everything works as expected under normal conditions.
- Try to break it.

> *Enter blank or random characters. Try it on differently sized devices. Try it on a device with a Bluetooth keyboard.*

Know that software always has bugs, and don't waste time striving for perfection.

Open a GitHub Pull Request

When your code is ready for review:

- Commit it to your feature branch
- Open a pull request on GitHub (or get a commit link)

If You Get Stuck

If you can't figure out the answer to a problem, reading and writing are your best friends. Writing forces you to list your assumptions, describ your expectations, and clarify your ideas.

- Write **what you expect** your code to do. Use the debugging tools to determine **what's actually happening**. Describe **how they differ**.

> *See **Getting Help on Stack Overflow** for advice on writing good questions.*

- Read the documentation related to the areas where your code doesn't do what you expect. Use forums like Stack Overflow and the Apple Developer Forums to research your issue.

> *For example, if you get an error like "Terminating app due to uncaught exception 'NSInvalidArgumentException', reason: '-[__NSCFString appendString:]: nil argument'" you should search for that error on Stack Overflow and review **the** `appendString:` **documentation**.*

Often, reading and writing about your issue will lead you to a solution. Use **Bloc Questions** to post your question on Stack Overflow. Send a link to your mentor too.

Repeat

These are the basic steps for completing one user story. Continue with the next story from the **Pick a User Story** section until they're completed.

Expectations

Plan to spend anywhere from **40 to 80 hours** on each project and complete **at least three** of them in order to graduate.

Pair Programming

Many mobile developers work alone and perhaps you will too someday, but today is not that day. Your mentor will play a large role in the development of your projects. The focus of your meetings from here on out should be implementation and programming. Working together, you and your mentor will code side-by-side as the project progresses.

This is what's known as **pair-programming**. One developer, the *driver*, writes the code while the other, an *observer* reviews each line as it is typed. The role of driver and observer will reverse among you and your mentor as needed.

Pull Requests

Maintain a clean `master` branch within your repository. It should ideally possess functional, crash-free code that accomplishes its purpose an needs no modification. Employ branching for when new work is required. For instance, after first cloning the repository to your local machine you may decide to add a new search view controller to the project. Create a branch for this exact purpose, like `search-view-controller`. This is called a *feature branch*.

When you have completed and tested this branch, submit a **pull request**. Your mentor will then review your request to merge `search-view-controller` into `master`. This allows your mentor to view your changes, test your code and give you feedback.

Once your mentor is satisfied, they will merge the branch into `master` and work may proceed. Create a new branch off master for your next task and repeat until the project is completed.

Read GitHub's **introduction to pull requests** (using the "Shared repository model"). Read this **article on the pros and cons of pull requests** and ask your mentor any questions you may have regarding the expected work flow of projects.

In your Blocstagram project, make a change and submit it via a pull request. Ask your mentor to review it. Then, pick a project, get to work and have fun!

assignment incomplete

hello@bloc.io
Considering enrolling? (404) 480-2562
Partnership / Corporate Inquiries? (650) 741-5682