```
+
+ - (instancetype) initWithCoder:(NSCoder *)aDecoder {
+     self = [super init];
+
+     if (self) {
+         self.idNumber = [aDecoder decodeObjectForKey:NSStringFromSelector(@selector(idNumber))];
+         self.text = [aDecoder decodeObjectForKey:NSStringFromSelector(@selector(text))];
+         self.from = [aDecoder decodeObjectForKey:NSStringFromSelector(@selector(from))];
+     }
+
+     return self;
+ }
+
+ - (void) encodeWithCoder:(NSCoder *)aCoder {
+     [aCoder encodeObject:self.idNumber forKey:NSStringFromSelector(@selector(idNumber))];
+     [aCoder encodeObject:self.text forKey:NSStringFromSelector(@selector(text))];
+     [aCoder encodeObject:self.from forKey:NSStringFromSelector(@selector(from))];
+ }
```

BLCMedia.h

```
- @interface BLCMedia : NSObject
+ @interface BLCMedia : NSObject <NSCoding>
```

BLCMedia.m

```
+ #pragma mark - NSCoding
+
+ - (instancetype) initWithCoder:(NSCoder *)aDecoder {
+     self = [super init];
+
+     if (self) {
+         self.idNumber = [aDecoder decodeObjectForKey:NSStringFromSelector(@selector(idNumber))];
+         self.user = [aDecoder decodeObjectForKey:NSStringFromSelector(@selector(user))];
+         self.mediaURL = [aDecoder decodeObjectForKey:NSStringFromSelector(@selector(mediaURL))];
+         self.image = [aDecoder decodeObjectForKey:NSStringFromSelector(@selector(image))];
+         self.caption = [aDecoder decodeObjectForKey:NSStringFromSelector(@selector(caption))];
+         self.comments = [aDecoder decodeObjectForKey:NSStringFromSelector(@selector(comments))];
+     }
+
+     return self;
+ }
+
+ - (void) encodeWithCoder:(NSCoder *)aCoder {
+     [aCoder encodeObject:self.idNumber forKey:NSStringFromSelector(@selector(idNumber))];
+     [aCoder encodeObject:self.user forKey:NSStringFromSelector(@selector(user))];
+     [aCoder encodeObject:self.mediaURL forKey:NSStringFromSelector(@selector(mediaURL))];
+     [aCoder encodeObject:self.image forKey:NSStringFromSelector(@selector(image))];
+     [aCoder encodeObject:self.caption forKey:NSStringFromSelector(@selector(caption))];
+     [aCoder encodeObject:self.comments forKey:NSStringFromSelector(@selector(comments))];
+ }
```

> *There are a few third-party libraries that try to break the tedium, including Nick Lockwood's **AutoCoding** and GitHub's **Mantle**. We don't use them in the Bloc curriculum, but they're worth looking at.*

## Adding **`NSKeyedArchiver`** to actually save and read the files

Now that the objects can be archived and unarchived, let's update **`BLCDataSource`** to save the file to disk and check for it at launch.

Add a method to create the full path to a file given a filename:

```
+     NSArray *paths = NSSearchPathForDirectoriesInDomains(NSCachesDirectory, NSUserDomainMask, YES);
+     NSString *documentsDirectory = [paths firstObject];
+     NSString *dataPath = [documentsDirectory stringByAppendingPathComponent:filename];
+     return dataPath;
+ }
```

This code will create a string containing an absolute path to the user's documents directory (like **`/somedir/someotherdir/filename`**).

We'll need to write to the file when we get new data. Let's do that at the end of **`parseDataFromFeedDictionary:fromRequestWithParameters:`**

BLCDataSource.m

```
        self.mediaItems = tmpMediaItems;
        [self didChangeValueForKey:@"mediaItems"];
    }
+
+     if (tmpMediaItems.count > 0) {
+         // Write the changes to disk
+         dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
+             NSUInteger numberOfItemsToSave = MIN(self.mediaItems.count, 50);
+             NSArray *mediaItemsToSave = [self.mediaItems subarrayWithRange:NSMakeRange(0, numberOfItemsToSave)];
+
+             NSString *fullPath = [self pathForFilename:NSStringFromSelector(@selector(mediaItems))];
+             NSData *mediaItemData = [NSKeyedArchiver archivedDataWithRootObject:mediaItemsToSave];
+
+             NSError *dataError;
+             BOOL wroteSuccessfully = [mediaItemData writeToFile:fullPath options:NSDataWritingAtomic | NSDataWritingFileProtectionC
+
+             if (!wroteSuccessfully) {
+                 NSLog(@"Couldn't write file: %@", dataError);
+             }
+         });
+
+     }
 }
```

Just like connecting to the Internet, reading or writing to disk can be slow. It's best to **`dispatch_async`** onto a background queue to do the file work you need.

In this code, after we're on a background queue, we make an **`NSArray`** containing the first 50 items (so we don't flood the user's hard drive). We convert this array into an **`NSData`** and save it to disk.

> *When we save it to disk, we pass it two options:* **`NSDataWritingAtomic`** *and* **`NSDataWritingFileProtectionCompleteUnlessOpen`**. *You should always pass these two options unless you have a compelling reason not to.*
>
> **`NSDataWritingAtomic`** *ensures a complete file is saved. Without it, we might corrupt our file if the app crashes while writing to disk.*
>
> **`NSDataWritingFileProtectionCompleteUnlessOpen`** *encrypts the data. This helps protect the user's privacy.*

Finally, we'll need to read the file at launch:

BLCDataSource.m

```
            [self registerForAccessTokenNotification];
        } else {
-           [self populateDataWithParameters:nil completionHandler:nil];
+           dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
+               NSString *fullPath = [self pathForFilename:NSStringFromSelector(@selector(mediaItems))];
+               NSArray *storedMediaItems = [NSKeyedUnarchiver unarchiveObjectWithFile:fullPath];
+
+               dispatch_async(dispatch_get_main_queue(), ^{
+                   if (storedMediaItems.count > 0) {
+                       NSMutableArray *mutableMediaItems = [storedMediaItems mutableCopy];
+
+                       [self willChangeValueForKey:@"mediaItems"];
+                       self.mediaItems = mutableMediaItems;
+                       [self didChangeValueForKey:@"mediaItems"];
+                   } else {
+                       [self populateDataWithParameters:nil completionHandler:nil];
+                   }
+               });
+           });
        }
```

This read-code is essentially the inverse of the write-code. It tries to find the file at the path and convert it into an array. If it finds an array of a least one item, it displays it immediately. (We make a **`mutableCopy`** since the copy stored to disk is immutable.) If not, it gets the initial data from the server.

Run your app. Then run it again. On the second launch, the images should appear immediately.



Add code to implement the following logic:

- On app launch, if cached images are found on disk and displayed, try to fetch newer content from the Instagram API.

This will avoid forcing the user to pull-to-refresh each time they launch the app.

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

Terminal

```
$ git add .
$ git commit -m 'Persist access token and Instagram content'
$ git checkout master
$ git merge persist-data
$ 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

Roadmap · Previous Checkpoint
Jump to Submission · Next Checkpoir

# Full Screen Photo Viewing and Sharing


iOS apps come alive when users can interact with all of the available content. When developers go the extra mile to make content interactive and shareable, users notice.

In this checkpoint, we'll add functionality so that you can tap on an Instagram photo to view it in full screen. We'll also enable the ability to share and save photos. This feature is noticeably absent from the official Instagram client.

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

Terminal

```
$ git checkout -b full-screen-images
```

## Cell Delegation

We'll start by adding a delegate method to **`BLCMediaTableViewCell`** which will inform the cell's controller when the user taps on the image.

We've used delegates and protocols before. This pattern should look familiar.

In the header file, we simply define the protocol:

BLCMediaTableViewCell.h

```
- @class BLCMedia;
+ @class BLCMedia, BLCMediaTableViewCell;
+
+ @protocol BLCMediaTableViewCellDelegate <NSObject>
+
+ - (void) cell:(BLCMediaTableViewCell *)cell didTapImageView:(UIImageView *)imageView;
+
+ @end

@interface BLCMediaTableViewCell : UITableViewCell

@property (nonatomic, strong) BLCMedia *mediaItem;
+ @property (nonatomic, weak) id <BLCMediaTableViewCellDelegate> delegate;

+ (CGFloat) heightForMediaItem:(BLCMedia *)mediaItem width:(CGFloat)width;
```

In the implementation, we'll add a tap gesture recognizer, and call the delegate if it's tapped.

Declare that we conform to the gesture recognizer delegate protocol:

BLCMediaTableViewCell.m

```
- @interface BLCMediaTableViewCell ()
+ @interface BLCMediaTableViewCell () <UIGestureRecognizerDelegate>
```

Add a property for the gesture recognizer:

BLCMediaTableViewCell.m

```
@property (nonatomic, strong) NSLayoutConstraint *usernameAndCaptionLabelHeightConstraint;
@property (nonatomic, strong) NSLayoutConstraint *commentLabelHeightConstraint;

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

Add the gesture recognizer to the image view:

```
    self = [super initWithStyle:style reuseIdentifier:reuseIdentifier];
    if (self) {
        self.mediaImageView = [[UIImageView alloc] init];
+         self.mediaImageView.userInteractionEnabled = YES;
+
+         self.tapGestureRecognizer = [[UITapGestureRecognizer alloc] initWithTarget:self action:@selector(tapFired:)];
+         self.tapGestureRecognizer.delegate = self;
+         [self.mediaImageView addGestureRecognizer:self.tapGestureRecognizer];

        self.usernameAndCaptionLabel = [[UILabel alloc] init];
        self.usernameAndCaptionLabel.numberOfLines = 0;
```

Add the target method, which will call **`cell:didTapImageView:`**.

BLCMediaTableViewCell.m

```
+
+ - (void) tapFired:(UITapGestureRecognizer *)sender {
+     [self.delegate cell:self didTapImageView:self.mediaImageView];
+ }
```

If the user swipes the cell to show the delete button, and then taps on the cell, the expected behavior is to hide the delete button, not to zoom in on the image. To accomplish this, we'll make sure the gesture recognizer only fires while the cell isn't in editing mode:

```
+ #pragma mark - UIGestureRecognizerDelegate
+
+ - (BOOL) gestureRecognizer:(UIGestureRecognizer *)gestureRecognizer shouldReceiveTouch:(UITouch *)touch {
+     return self.isEditing == NO;
+ }
```

Finally, to avoid any issues redrawing the cells if the user rotates the device while the full-screen image is showing, remove the calculation of the image height constraint from **`setMediaItem:`**...

BLCMediaTableViewCell.m

```
    self.usernameAndCaptionLabel.attributedText = [self usernameAndCaptionString];
    self.commentLabel.attributedText = [self commentString];

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

And add it to **`layoutSubviews`**:

BLCMediaTableViewCell.m

```
    self.usernameAndCaptionLabelHeightConstraint.constant = usernameLabelSize.height + 20;
    self.commentLabelHeightConstraint.constant = commentLabelSize.height + 20;

+     if (_mediaItem.image) {
+         self.imageHeightConstraint.constant = self.mediaItem.image.size.height / self.mediaItem.image.size.width * CGRectGetWidth(s
+     } else {
+         self.imageHeightConstraint.constant = 0;
+     }
+
    // Hide the line between cells
    self.separatorInset = UIEdgeInsetsMake(0, 0, 0, CGRectGetWidth(self.bounds));
```

# Creating a Full-Screen Image Viewer

The next piece we need is a view controller to handle displaying a **`BLCMedia`** item in full-screen. Create a new **`UIViewController`** subclass called **`BLCMediaFullScreenViewController`**.

Here's what the header file looks like:

BLCMediaFullScreenViewController.h

```
+
+ @interface BLCMediaFullScreenViewController : UIViewController
+
+ @property (nonatomic, strong) UIScrollView *scrollView;
+ @property (nonatomic, strong) UIImageView *imageView;
+
+ - (instancetype) initWithMedia:(BLCMedia *)media;
+
+ - (void) centerScrollView;
+
+ @end
```

The scroll view and image view will be explained in a bit. Aside from that, this is just like other view controllers you've made - you have a custom initializer; in this one, you will pass it a **`BLCMedia`** object for it to display.

Let's look at the implementation. We'll start by importing **`BLCMedia`** and creating a property to hang on to it. We'll also declare that this class conforms to the **`UIScrollViewDelegate`** protocol.

BLCMediaFullScreenViewController.m

```
 #import "BLCMediaFullScreenViewController.h"
+ #import "BLCMedia.h"

- @interface BLCMediaFullScreenViewController ()
+ @interface BLCMediaFullScreenViewController () <UIScrollViewDelegate>

+ @property (nonatomic, strong) BLCMedia *media;

 @end
```

In the initializer, simply store the media item for later use:

BLCMediaFullScreenViewController.m

```
 @implementation BLCMediaFullScreenViewController

+ - (instancetype) initWithMedia:(BLCMedia *)media {
+     self = [super init];
+
+     if (self) {
+         self.media = media;
+     }
+
+     return self;
+ }
```

For actually displaying the image, we'll use a **`UIScrollView`**. Scroll views don't just slide content around on the screen - they also make it easy to zoom in and out.

Here's a summary of the work we'll need to do from the **`UIScrollView` class reference**:

> *A scroll view also handles zooming and panning of content. As the user makes a pinch-in or pinch-out gesture, the scroll view adjusts the offset and the scale of the content. When the gesture ends, the object managing the content view should should update subviews of the content as necessary.*
>
> *The* **`UIScrollView`** *class can have a delegate that must adopt the* **`UIScrollViewDelegate`** *protocol. For zooming and panning to work, the delegate must implement both* **`viewForZoomingInScrollView:`** *and* **`scrollViewDidEndZooming:withView:atScale:`***; in addition, the maximum (*`maximumZoomScale`*) and minimum (*`minimumZoomScale`*) zoom scale must be different.*

In **`viewDidLoad`**, we'll set up the scroll view and image view:

```
+ - (void) viewDidLoad {
+     [super viewDidLoad];
+
+     self.scrollView = [UIScrollView new];
+     self.scrollView.delegate = self;
+     self.scrollView.backgroundColor = [UIColor whiteColor];
+
+     [self.view addSubview:self.scrollView];
+
+     self.imageView = [UIImageView new];
+     self.imageView.image = self.media.image;
+
+     [self.scrollView addSubview:self.imageView];
+     self.scrollView.contentSize = self.media.image.size;
+ }
```

Most of this should look familiar. We create and configure a scroll view, and add it as the only subview of **`self.view`**. We then create an image view, set the image, and add it as a subview of the scroll view.

You haven't seen a scroll view's **`contentSize`** before: this property represents the size of the content view, which is the content being scrolled around. In our case, we're simply scrolling around an image, so we'll pass in its size.

You'll notice we haven't set either view's **`frame`** yet. We also haven't set **`minimumZoomScale`** and **`maximumZoomScale`** yet, as mentioned in the documentation. We'll take care of that in **`viewWillLayoutSubviews`**:

BLCMediaFullScreenViewController.m

```
+ - (void) viewWillLayoutSubviews {
+     [super viewWillLayoutSubviews];
+     self.scrollView.frame = self.view.bounds;
+
+     CGSize scrollViewFrameSize = self.scrollView.frame.size;
+     CGSize scrollViewContentSize = self.scrollView.contentSize;
+
+     CGFloat scaleWidth = scrollViewFrameSize.width / scrollViewContentSize.width;
+     CGFloat scaleHeight = scrollViewFrameSize.height / scrollViewContentSize.height;
+     CGFloat minScale = MIN(scaleWidth, scaleHeight);
+
+     self.scrollView.minimumZoomScale = minScale;
+     self.scrollView.maximumZoomScale = 1;
+ }
```

First, the scroll view's **`frame`** is set to the view's **`bounds`**. This way, the scroll view will always take up all of the view's space.

Then we look at two ratios:

1. the ratio of the scroll view's width to the image's width
2. the ratio of the scroll view's height to the image's height

Whichever is smaller will become our **`minimumZoomScale`**. (This prevents the user from pinching the image so small that there's wasted screer space.)

**`maximumZoomScale`** will always be **`1`** (representing 100%). We could make this bigger, but then the image would just start to get pixelated if the user zooms in too much.

We'll now implement **`centerScrollView`**. If the image is zoomed down so that it doesn't fill the full scroll view, this method will center the image on that axis.

For example, if the image is square (Instagram images always are), the app is being run on a device that's taller than it is wide, and the user has it zoomed to its minimum size, this method ensures equal blank space on the top and bottom:


Here's the implementation:

BLCMediaFullScreenViewController.m

```
+ - (void)centerScrollView {
+     [self.imageView sizeToFit];
+
+     CGSize boundsSize = self.scrollView.bounds.size;
+     CGRect contentsFrame = self.imageView.frame;
+
+     if (contentsFrame.size.width < boundsSize.width) {
+         contentsFrame.origin.x = (boundsSize.width - CGRectGetWidth(contentsFrame)) / 2;
+     } else {
+         contentsFrame.origin.x = 0;
+     }
+
+     if (contentsFrame.size.height < boundsSize.height) {
+         contentsFrame.origin.y = (boundsSize.height - CGRectGetHeight(contentsFrame)) / 2;
+     } else {
+         contentsFrame.origin.y = 0;
+     }
+
+     self.imageView.frame = contentsFrame;
+ }
```

We'll now implement two scroll view delegate methods, one of which will call **`centerScrollView`**:

BLCMediaFullScreenViewController.m

```
+
+ - (UIView*)viewForZoomingInScrollView:(UIScrollView *)scrollView {
+     return self.imageView;
+ }
+
+ - (void)scrollViewDidZoom:(UIScrollView *)scrollView {
+     [self centerScrollView];
+ }
```

The first method tells the scroll view which view to zoom in and out on; the latter calls **`centerScrollView`** when the user has changed the zoom level.

Also, let's make sure the image starts out centered:

BLCMediaFullScreenViewController.m

```
+ - (void) viewWillAppear:(BOOL)animated {
+     [super viewWillAppear:animated];
+
+     [self centerScrollView];
+ }
```

This view controller is basically complete now, but let's add two finishing touches (pun intended): tap and double-tap events.

# Tap and Double-Tap Events

Let's allow the user to dismiss the view controller by tapping on it, and to change the zoom level by double-tapping.

> *Any time you implement a pinch gesture, it's a good idea to implement a double-tap gesture too. Not only is it handy, but it's very helpful to disabled users who may be unable to pinch. (For example, some users are missing fingers, and paraplegic users may be using a stylus in their mouth.)*

Add properties for the gesture recognizers:

BLCMediaFullScreenViewController.m

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

Initialize them in **`viewDidLoad`**:

BLCMediaFullScreenViewController.m

```
    self.scrollView.contentSize = self.media.image.size;
+
+     self.tap = [[UITapGestureRecognizer alloc] initWithTarget:self action:@selector(tapFired:)];
+
+     self.doubleTap = [[UITapGestureRecognizer alloc] initWithTarget:self action:@selector(doubleTapFired:)];
+     self.doubleTap.numberOfTapsRequired = 2;
+
+     [self.tap requireGestureRecognizerToFail:self.doubleTap];
+
+     [self.scrollView addGestureRecognizer:self.tap];
+     [self.scrollView addGestureRecognizer:self.doubleTap];
 }
```

Two things may be new to you here:

- **`numberOfTapsRequired`** allows a tap gesture recognizer to require more than one tap to fire

Without this line, it would be impossible to double-tap because the single tap gesture recognizer would fire before the user had a chanc to tap twice.

When the user single-taps, simply dismiss the view controller:

BLCMediaFullScreenViewController.m

```
+ #pragma mark - Gesture Recognizers
+
+ - (void) tapFired:(UITapGestureRecognizer *)sender {
+     [self dismissViewControllerAnimated:YES completion:nil];
+ }
```

When the user double-taps, adjust the zoom level:

BLCMediaFullScreenViewController.m

```
+ - (void) doubleTapFired:(UITapGestureRecognizer *)sender {
+     if (self.scrollView.zoomScale == self.scrollView.minimumZoomScale) {
+         CGPoint locationPoint = [sender locationInView:self.imageView];
+
+         CGSize scrollViewSize = self.scrollView.bounds.size;
+
+         CGFloat width = scrollViewSize.width / self.scrollView.maximumZoomScale;
+         CGFloat height = scrollViewSize.height / self.scrollView.maximumZoomScale;
+         CGFloat x = locationPoint.x - (width / 2);
+         CGFloat y = locationPoint.y - (height / 2);
+
+         [self.scrollView zoomToRect:CGRectMake(x, y, width, height) animated:YES];
+     } else {
+         [self.scrollView setZoomScale:self.scrollView.minimumZoomScale animated:YES];
+     }
+ }
```

If the current zoom scale is already as small as it can be, double-tapping will zoom in. This works by calculating a rectangle using the user's finger as a center point, and telling the scroll view to zoom in on that rectangle.

If the current zoom scale is larger, then zoom out to the minimum scale.

We're done building the view controller. Now let's wire it up to the cell.

# Presenting the View Controller

When the user taps the image, our table view controller will need to present the full screen image controller.

Let's get started wiring that up; it won't take long.

Import the new view controller we just made:

BLCImagesTableViewController.m

```
 #import "BLCComment.h"
 #import "BLCMediaTableViewCell.h"
+ #import "BLCMediaFullScreenViewController.h"
```

Indicate that we conform to the new protocol we made at the beginning of this checkpoint:

BLCImagesTableViewController.m

```
- @interface BLCImagesTableViewController ()
+ @interface BLCImagesTableViewController () <BLCMediaTableViewCellDelegate>
```

BLCImagesTableViewController.m

```
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    BLCMediaTableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:@"mediaCell" forIndexPath:indexPath];
+     cell.delegate = self;
    cell.mediaItem = [BLCDataSource sharedInstance].mediaItems[indexPath.row];
    return cell;
}
```

Finally, we need to implement the delegate method:

BLCImagesTableViewController.m

```
+ #pragma mark - BLCMediaTableViewCellDelegate
+
+ - (void) cell:(BLCMediaTableViewCell *)cell didTapImageView:(UIImageView *)imageView {
+     BLCMediaFullScreenViewController *fullScreenVC = [[BLCMediaFullScreenViewController alloc] initWithMedia:cell.mediaItem];
+
+     [self presentViewController:fullScreenVC animated:YES completion:nil];
+ }
```

Run the app. You should now be able to:

- tap on an image to view it in full screen
- tap on the full-screen image to dismiss it
- double-tap on the full-screen image to zoom in and out
- pinch the full-screen image to zoom in and out

You'll notice the full-screen image slides up and down from the bottom of the screen.

Let's make that animation a bit cooler by having the tapped image zoom in and out.

# Custom View Controller Animations

We'll make a new type of object called a *transition coordinator*. It sounds complicated, but the implementation won't be too hard. It is responsible for coordinating the animation between two view controllers.

Create a new **`NSObject`** subclass called **`BLCMediaFullScreenAnimator`**.

Here's what the interface file should look like:

BLCMediaFullScreenAnimator.h

```
+ @interface BLCMediaFullScreenAnimator : NSObject <UIViewControllerAnimatedTransitioning>
+
+ @property (nonatomic, assign) BOOL presenting;
+ @property (nonatomic, weak) UIImageView *cellImageView;
+
+ @end
```

We have two properties. The first, **`presenting`**, will let us know if the animation is a presenting animation (if not, it's a dismissing animation). The second, **`cellImageView`**, will reference the image view from the media table view cell (the image view the user taps on).

In the implementation file, we'll import the full screen view controller:

BLCMediaFullScreenAnimator.m

```
+ #import "BLCMediaFullScreenViewController.h"
```

BLCMediaFullScreenAnimator.m

```
+ - (NSTimeInterval)transitionDuration:(id <UIViewControllerContextTransitioning>)transitionContext {
+     return 0.2;
+ }
```

> *An* **`NSTimeInterval`** *can be any* **`double`** *number. It's always specified in seconds.*

The second method works generally like this:

- The method is passed a **transition context** object called **`transitionContext`**. This object gives us 3 crucial pieces of information:
  1. The view controller the user is leaving (**`UITransitionContextFromViewControllerKey`**)
  2. The view controller the user is going to (**`UITransitionContextToViewControllerKey`**)
  3. A **container view**, which contains the views of both view controllers during the animation
- We calculate (and specify) the starting and ending **`frame`** for the view controller that's moving.
- We animate the view controller using the frames we calculated.
- Finally, we inform the transition context when the animation is completed.

We'll do this in two separate ways, depending on whether **`self.presented`** is **`YES`** or **`NO`**.

Take a look the implementation and then we'll step through it:

BLCMediaFullScreenAnimator.m

```
+    UIViewController *fromViewController = [transitionContext viewControllerForKey:UITransitionContextFromViewControllerKey];
+    UIViewController *toViewController = [transitionContext viewControllerForKey:UITransitionContextToViewControllerKey];
+
+    if (self.presenting) {
+        BLCMediaFullScreenViewController *fullScreenVC = (BLCMediaFullScreenViewController *)toViewController;
+
+        fromViewController.view.userInteractionEnabled = NO;
+
+        [transitionContext.containerView addSubview:toViewController.view];
+
+        CGRect startFrame = [transitionContext.containerView convertRect:self.cellImageView.bounds fromView:self.cellImageView];
+        CGRect endFrame = fromViewController.view.frame;
+
+        toViewController.view.frame = startFrame;
+        fullScreenVC.imageView.frame = toViewController.view.bounds;
+
+        [UIView animateWithDuration:[self transitionDuration:transitionContext] animations:^{
+            fromViewController.view.tintAdjustmentMode = UIViewTintAdjustmentModeDimmed;
+
+            fullScreenVC.view.frame = endFrame;
+            [fullScreenVC centerScrollView];
+        } completion:^(BOOL finished) {
+            [transitionContext completeTransition:YES];
+        }];
+    }
+    else {
+        BLCMediaFullScreenViewController *fullScreenVC = (BLCMediaFullScreenViewController *)fromViewController;
+
+        CGRect endFrame = [transitionContext.containerView convertRect:self.cellImageView.bounds fromView:self.cellImageView];
+        CGRect imageStartFrame = [fullScreenVC.view convertRect:fullScreenVC.imageView.frame fromView:fullScreenVC.scrollView];
+        CGRect imageEndFrame = [transitionContext.containerView convertRect:endFrame toView:fullScreenVC.view];
+
+        imageEndFrame.origin.y = 0;
+
+        [fullScreenVC.view addSubview:fullScreenVC.imageView];
+        fullScreenVC.imageView.frame = imageStartFrame;
+        fullScreenVC.imageView.autoresizingMask = UIViewAutoresizingFlexibleBottomMargin;
+
+        toViewController.view.userInteractionEnabled = YES;
+
+        [UIView animateWithDuration:[self transitionDuration:transitionContext] animations:^{
+            fullScreenVC.view.frame = endFrame;
+            fullScreenVC.imageView.frame = imageEndFrame;
+
+            toViewController.view.tintAdjustmentMode = UIViewTintAdjustmentModeAutomatic;
+        } completion:^(BOOL finished) {
+            [transitionContext completeTransition:YES];
+        }];
+    }
+ }
```

Let's first look at this code as if **`self.presenting`** is **`YES`** (which will be the case after a user taps an image):

**`fromViewController`** will point to the table view controller.

**`toViewController`** will point to the full screen image view controller.

**`self.presenting == YES`**, so we'll go in the first conditional block.

We know that **`toViewController`** is the full screen view controller, so we make a new variable name, **`fullScreenVC`**, to refer to it. (This is calle *casting*; you've used it before.)

We disable user interaction on the table view controller, so the users can't scroll while the animation is happening.

We add the views of both view controllers to the transition context's container view.

We'll now start calculating frames, and then perform the animation.

user just tapped on. Remember that a **`frame`** describes a view's position and size *with respect to its superview*.

**`fullScreenVC`**'s view is in the coordinate system of **`transitionContext.containerView`**, but **`self.cellImageView`** is in a different coordinate system (it's in a cell, in a table, in another view controller). To convert, we call the method **`convertRect:fromView`**, which converts a view's **`frame`** or **`bounds`** from one view's coordinate system into the receiver's.

**`startFrame`** is now the frame of the image view the user just tapped on, except converted into the container view's coordinate system.

**`endFrame`** is easier: we want the full screen view to use the same position and sizing as the regular view controller, so we just copy its frame.

Now that we've calculated the necessary frames, we can complete the animation.

In the most general sense, animations in iOS look like this:

```
/* Configure how stuff looks when the animation starts */

[UIView animateWithDuration:/* How long the animation takes */
                 animations:^{
                     /* Configure how stuff looks when the animation ends */
                 }
                 completion:^(BOOL finished) {
                     /* Do whatever you want when the animation has completed */
                 }];
```

You can use this simple pattern to create basic animations. In this code, we only animate a view's **`frame`** properties. But it's possible to animate a few other properties, including **`alpha`** and **`backgroundColor`**.

So, our presentation animation looks like this:

```
toViewController.view.frame = startFrame;
fullScreenVC.imageView.frame = toViewController.view.bounds;

[UIView animateWithDuration:[self transitionDuration:transitionContext] animations:^{
    fromViewController.view.tintAdjustmentMode = UIViewTintAdjustmentModeDimmed;

    fullScreenVC.view.frame = endFrame;
    [fullScreenVC centerScrollView];
} completion:^(BOOL finished) {
    [transitionContext completeTransition:YES];
}];
```

We set the full screen view controller's frame to be directly over the tapped image.

We set the image view's frame to fill the frame completely.

We animate the frame to **`endFrame`**, which fills the entire screen. We also center the image if necessary.

Finally, when the animation is complete, we inform the transition context that the transition has completed.

The code for dismissing the view controller (when **`self.presenting == NO`** is similar, but backwards, so we won't rehash it here.

Now all we need to do is tell iOS to use our new animator.

# Hooking up the Animator

Hooking the animator up to our view is easy.

Open up the table view controller and import the animator:

BLCImagesTableViewController.m

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

Declare that the table controller conforms to the **`UIViewControllerTransitioningDelegate`** protocol, and add a property to track which view was tapped most recently:

BLCImagesTableViewController.m

```
- @interface BLCImagesTableViewController () <BLCMediaTableViewCellDelegate>
+ @interface BLCImagesTableViewController () <BLCMediaTableViewCellDelegate, UIViewControllerTransitioningDelegate>
+
+ @property (nonatomic, weak) UIImageView *lastTappedImageView;
```

We need to implement the transitioning delegate methods. These methods simply vend an animator object to iOS. Note that when we're presenting, we set **`presenting`** to **`YES`**, but otherwise we don't.

BLCImagesTableViewController.m

```
+ #pragma mark - UIViewControllerTransitioningDelegate
+
+ - (id<UIViewControllerAnimatedTransitioning>)animationControllerForPresentedController:(UIViewController *)presented
+                                                                    presentingController:(UIViewController *)presenting
+                                                                        sourceController:(UIViewController *)source {
+
+     BLCMediaFullScreenAnimator *animator = [BLCMediaFullScreenAnimator new];
+     animator.presenting = YES;
+     animator.cellImageView = self.lastTappedImageView;
+     return animator;
+ }
+
+ - (id<UIViewControllerAnimatedTransitioning>)animationControllerForDismissedController:(UIViewController *)dismissed {
+     BLCMediaFullScreenAnimator *animator = [BLCMediaFullScreenAnimator new];
+     animator.cellImageView = self.lastTappedImageView;
+     return animator;
+ }
```

Finally, we just need to set **`lastTappedImageView`** when an image view is tapped, and let iOS know that this transition uses a delegate:

BLCImagesTableViewController.m

```
 - (void) cell:(BLCMediaTableViewCell *)cell didTapImageView:(UIImageView *)imageView {
+     self.lastTappedImageView = imageView;
+
     BLCMediaFullScreenViewController *fullScreenVC = [[BLCMediaFullScreenViewController alloc] initWithMedia:cell.mediaItem];

+     fullScreenVC.transitioningDelegate = self;
+     fullScreenVC.modalPresentationStyle = UIModalPresentationCustom;
+
     [self presentViewController:fullScreenVC animated:YES completion:nil];
 }
```

Run the app again. You should now be able to see the full screen effect shown in the screenshot at the beginning of this checkpoint.

# Sharing is Caring

Now that we have a delegate protocol set up for **`BLCMediaTableViewCell`**, let's throw in an easy feature to take advantage of it: image sharing

There are many ways to share content on iOS. By far the simplest approach is a built-in class called **`UIActivityViewController`**, which provides automated sharing depending on which options you provide and which apps the user has installed.

Our goal is to trigger a share sheet if the user long-presses on an image.

BLCMediaTableViewCell.h

```
@protocol BLCMediaTableViewCellDelegate <NSObject>

- (void) cell:(BLCMediaTableViewCell *)cell didTapImageView:(UIImageView *)imageView;
+ - (void) cell:(BLCMediaTableViewCell *)cell didLongPressImageView:(UIImageView *)imageView;

@end
```

Add a long press recognizer property to your cell:

BLCMediaTableViewCell.m

```
@property (nonatomic, strong) UITapGestureRecognizer *tapGestureRecognizer;
+ @property (nonatomic, strong) UILongPressGestureRecognizer *longPressGestureRecognizer;

@end
```

Initialize it and add it to the image view:

BLCMediaTableViewCell.m

```
        [self.mediaImageView addGestureRecognizer:self.tapGestureRecognizer];

+         self.longPressGestureRecognizer = [[UILongPressGestureRecognizer alloc] initWithTarget:self action:@selector(longPressFired
+         self.longPressGestureRecognizer.delegate = self;
+         [self.mediaImageView addGestureRecognizer:self.longPressGestureRecognizer];
+
        self.usernameAndCaptionLabel = [[UILabel alloc] init];
```

When the gesture recognizer fires, inform the delegate:

BLCMediaTableViewCell.m

```
    [self.delegate cell:self didTapImageView:self.mediaImageView];
}

+ - (void) longPressFired:(UILongPressGestureRecognizer *)sender {
+     if (sender.state == UIGestureRecognizerStateBegan) {
+         [self.delegate cell:self didLongPressImageView:self.mediaImageView];
+     }
+ }
```

We make sure that **state** is **UIGestureRecognizerStateBegan**. We could alternatively check for **UIGestureRecognizerStateRecognized**, but then the method wouldn't get called until the user lifts their finger. (And if we don't check at all, the delegate method will get called twice: onc when the recognizer begins, and again when the user lifts their finger.)

Finally, implement the delegate method in the images table view controller:

BLCImagesTableViewController.m

```
}

+ - (void) cell:(BLCMediaTableViewCell *)cell didLongPressImageView:(UIImageView *)imageView {
+     NSMutableArray *itemsToShare = [NSMutableArray array];
+
+     if (cell.mediaItem.caption.length > 0) {
+         [itemsToShare addObject:cell.mediaItem.caption];
+     }
+
+     if (cell.mediaItem.image) {
+         [itemsToShare addObject:cell.mediaItem.image];
+     }
+
+     if (itemsToShare.count > 0) {
+         UIActivityViewController *activityVC = [[UIActivityViewController alloc] initWithActivityItems:itemsToShare applicationActi
+         [self presentViewController:activityVC animated:YES completion:nil];
+     }
+ }
```

This will share an image (and a caption if there is one). **`UIActivityViewController`** is passed an array of items to share, and then it's presented.

**`UIActivityViewController`** takes care of everything else. Wasn't that easy?

# Recap

In this checkpoint you reused some skills you'd learned before:

- Creating gesture recognizers
- Creating delegate protocols

You also learned a lot more about scroll views.

Additionally, you learned more about the view controller class, including how to present view controllers and how to define your own custom transitions.

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

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

Play around with some of the code in this checkpoint:

- Experiment with different animation durations in the animator
- Experiment with different minimum and maximum zoom settings in the full screen view controller
- Look in the **`UIView`** class reference at the other methods that begin with **`animateWithDuration:...`**. Try using some others in the animator.

Finally, add a button to the upper-right corner of the full screen view controller that says "Share". When it's tapped, bring up the activity view controller.

Try to accomplish this without duplicating any code.

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

Terminal

```
$ git commit -m 'Full screen images and sharing'
$ git checkout master
$ git merge full-screen-images
$ git push
```

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

assignment completed

Roadmap · Previous Checkpoint Jump to Submission · Next Checkpoir

# 3rd Party Networking Library

> *"We can't help everyone, but everyone can help someone."*
>
> *- Ronald Reagan, American actor and politician*

In a prior checkpoint, you installed the **`UICKeyChainStore`** library using CocoaPods.

Now we'll install, configure, and use **`AFNetworking`**, a powerful networking library that makes receiving and sending data easy. It's one of the most common iOS libraries, and we'll use it to replace some code in our app.

In addition to abstracting away some of the complexities of networking, it's worth exploring **`AFNetworking`** simply because of its beauty. The code style and documentation is meticulously maintained. The use of protocols and the clear separation of concerns between classes provide exceptional modularity. This makes it easy to only use parts of **`AFNetworking`** without committing to all of it.

## Installing **`AFNetworking`**

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

Terminal

```
$ git checkout -b install-afnetworking
```

Open up your Podfile in your favorite text editor:

Terminal

```
$ open -a Textedit Podfile
```

Add **`AFNetworking`**:

Podfile

```
 pod 'UICKeyChainStore'
+ pod 'AFNetworking'
```

Save and close your Podfile, then run CocoaPods:

Terminal

```
$ pod install
Analyzing dependencies
Downloading dependencies
Installing AFNetworking (2.3.1)
Using UICKeyChainStore (1.0.5)
Generating Pods project
Integrating client project
```

# About **AFNetworking**'s classes

AFNetworking is a large, modular project that can do many different things.

Here are some of the classes we'll be using:

## HTTP Request Operation Manager

**AFHTTPRequestOperationManager** takes care of many of the tasks you need to do when communicating with a server, including:

- creation of **NSURLRequest** objects
- response serialization (converting from data that's received to objects like **NSDictionary** or **UIImage**)
- request serialization (converting from objects to outgoing data)
- security
- ordering and prioritizing network requests

## Response Serializer

Response serialization in iOS is the process of converting an **NSData** object to something more useful, usually **NSDictionary**, **NSArray**, or **UIImage**.

Most of the serialization we've done so far has been *response serialization*. For example, this code from **[BLCDataSource -downloadImageForMediaItem:]** is rudimentary response serialization:

```
NSURLResponse *response;
NSError *error;
NSData *imageData = [NSURLConnection sendSynchronousRequest:request returningResponse:&response error:&error];

if (imageData) {
    UIImage *image = [UIImage imageWithData:imageData];

    if (image) {
```

**AFNetworking** uses the **AFHTTPResponseSerializer** for general use, and provides two subclasses that we'll be interested in:

- **AFJSONResponseSerializer** validates and decodes JSON responses, producing an **NSArray** or **NSDictionary**.
- **AFImageResponseSerializer** validates and decodes image responses, producing a **UIImage**.

## Request Serializer

Request serialization is the opposite of response serialization: with request serialization, we're converting from foundation objects into a format that the server can read (either **NSString** or **NSData**).

For example, this code in our app is a basic example of request serialization:

```
NSMutableString *urlString = [NSMutableString stringWithFormat:@"https://api.instagram.com/v1/users/self/feed?access_token=%@", self.a

for (NSString *parameterName in parameters) {
    // for example, if dictionary contains {count: 50}, append `&count=50` to the URL
    [urlString appendFormat:@"&%@=%@", parameterName, parameters[parameterName]];
}

NSURL *url = [NSURL URLWithString:urlString];
```

The primary request serialization object is **AFHTTPRequestSerializer**; there are similar subclasses for serializing other types of data.

### HTTP Request Operation

Every request sent to a server is managed by one HTTP Request Operation (**`AFHTTPRequestOperation`**). Each operation has two completion blocks: success and failure. One or the other is guaranteed to be called when the operation finishes.

Success or failure of a request is determined by two things:

1. the response's status code (for example, **`200 OK`**, **`403 FORBIDDEN`** or **`404 NOT FOUND`**), and
2. the response's content type (for example, if you're expecting JSON but the server returns HTML).

HTTP Request Operation objects also handle **`dispatch_async`**ing onto a background queue before connecting, and again back to the main queue when finished.

**`AFHTTPRequestOperation`** objects are typically created and managed automatically by your **`AFHTTPRequestOperationManager`**, but they can also be used separately.

## Integrating **`AFNetworking`**

Open up **`BLCDataSource.m`**. We'll make all of our changes here.

Import **`AFNetworking`**:

BLCDataSource.m

```
#import "BLCLoginViewController.h"
#import <UICKeyChainStore.h>
+ #import <AFNetworking/AFNetworking.h>
```

Add a property for the manager:

BLCDataSource.m

```
@property (nonatomic, assign) BOOL isLoadingOlderItems;
@property (nonatomic, assign) BOOL thereAreNoMoreOlderMessages;

+ @property (nonatomic, strong) AFHTTPRequestOperationManager *instagramOperationManager;
+
@end
```

Initialize the operation manager in **`init`**:

BLCDataSource.m

```
    self = [super init];

    if (self) {
+       NSURL *baseURL = [NSURL URLWithString:@"https://api.instagram.com/v1/"];
+       self.instagramOperationManager = [[AFHTTPRequestOperationManager alloc] initWithBaseURL:baseURL];
+
+       AFJSONResponseSerializer *jsonSerializer = [AFJSONResponseSerializer serializer];
+
+       AFImageResponseSerializer *imageSerializer = [AFImageResponseSerializer serializer];
+       imageSerializer.imageScale = 1.0;
+
+       AFCompoundResponseSerializer *serializer = [AFCompoundResponseSerializer compoundSerializerWithResponseSerializers:@[jsonSe
+       self.instagramOperationManager.responseSerializer = serializer;
+
        self.accessToken = [UICKeyChainStore stringForKey:@"access token"];

        if (!self.accessToken) {
```

A few notes:

*Construction Using Relative Paths* in the `AFHTTPRequestOperationManager` **documentation**.

- Since some requests return JSON and others return images, we create a `AFCompoundResponseSerializer`. This saves us from having to specify for each request the type of object we want; AFNetworking will figure it out automatically.
- For the `AFImageResponseSerializer` instance, we set the `imageScale` to `1.0`. By default, the image response serializer will assume all images on Retina devices are double-resolution. Since this isn't true of the Instagram API, we just save them as normal.

Let's update `populateDataWithParameters:completionHandler:` to use the operation manager. We'll delete most of the current implementation and replace it with an implementation that uses `AFNetworking`:

BLCDataSource.m

```
        // only try to get the data if there's an access token

-       dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0), ^{
-           // do the network request in the background, so the UI doesn't lock up
-
-           NSMutableString *urlString = [NSMutableString stringWithFormat:@"https://api.instagram.com/v1/users/self/feed?access_to
-
-           for (NSString *parameterName in parameters) {
-               // for example, if dictionary contains {count: 50}, append `&count=50` to the URL
-               [urlString appendFormat:@"&%@=%@", parameterName, parameters[parameterName]];
-           }
-
-           NSURL *url = [NSURL URLWithString:urlString];
-
-           if (url) {
-               NSURLRequest *request = [NSURLRequest requestWithURL:url];
-
-               NSURLResponse *response;
-               NSError *webError;
-               NSData *responseData = [NSURLConnection sendSynchronousRequest:request returningResponse:&response error:&webError]
-
-               if (responseData) {
-                   NSError *jsonError;
-                   NSDictionary *feedDictionary = [NSJSONSerialization JSONObjectWithData:responseData options:0 error:&jsonError]
-
-                   if (feedDictionary) {
-                       dispatch_async(dispatch_get_main_queue(), ^{
-                           // done networking, go back on the main thread
-                           [self parseDataFromFeedDictionary:feedDictionary fromRequestWithParameters:parameters];
-                           if (completionHandler) {
-                               completionHandler(nil);
-                           }
-                       });
-                   } else if (completionHandler) {
-                       dispatch_async(dispatch_get_main_queue(), ^{
-                           completionHandler(jsonError);
-                       });
-                   }
-               } else if (completionHandler) {
-                   dispatch_async(dispatch_get_main_queue(), ^{
-                       completionHandler(webError);
-                   });
-               }
-           }
-       });
+       NSMutableDictionary *mutableParameters = [@{@"access_token": self.accessToken} mutableCopy];
+
+       [mutableParameters addEntriesFromDictionary:parameters];
+
+       [self.instagramOperationManager GET:@"users/self/feed"
+                       parameters:mutableParameters
+                          success:^(AFHTTPRequestOperation *operation, id responseObject) {
+                              if ([responseObject isKindOfClass:[NSDictionary class]]) {
+                                  [self parseDataFromFeedDictionary:responseObject fromRequestWithParameters:parameters];
+
+                                  if (completionHandler) {
+                                      completionHandler(nil);
+                                  }
+                              }
+                          } failure:^(AFHTTPRequestOperation *operation, NSError *error) {
+                              if (completionHandler) {
+                                  completionHandler(error);
+                              }
+                          }];
    }
}
```

In this implementation, we create a parameters dictionary for the access token, and add in any other parameters that might be passed in (like **`min_id`** or **`max_id`**). We tell the request operation manager to get the resource, and if it's successful, we send it to

BLCDataSource.m

```
- (void) downloadImageForMediaItem:(BLCMedia *)mediaItem {
    if (mediaItem.mediaURL && !mediaItem.image) {
-         dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
-             NSURLRequest *request = [NSURLRequest requestWithURL:mediaItem.mediaURL];
-
-             NSURLResponse *response;
-             NSError *error;
-             NSData *imageData = [NSURLConnection sendSynchronousRequest:request returningResponse:&response error:&error];
-
-             if (imageData) {
-                 UIImage *image = [UIImage imageWithData:imageData];
-
-                 if (image) {
-                     mediaItem.image = image;
-
-                     dispatch_async(dispatch_get_main_queue(), ^{
-                         NSMutableArray *mutableArrayWithKVO = [self mutableArrayValueForKey:@"mediaItems"];
-                         NSUInteger index = [mutableArrayWithKVO indexOfObject:mediaItem];
-                         [mutableArrayWithKVO replaceObjectAtIndex:index withObject:mediaItem];
-                     });
-                 }
-             } else {
-                 NSLog(@"Error downloading image: %@", error);
-             }
-         });
+         [self.instagramOperationManager GET:mediaItem.mediaURL.absoluteString
+                                  parameters:nil
+                                     success:^(AFHTTPRequestOperation *operation, id responseObject) {
+                                         if ([responseObject isKindOfClass:[UIImage class]]) {
+                                             mediaItem.image = responseObject;
+                                             NSMutableArray *mutableArrayWithKVO = [self mutableArrayValueForKey:@"mediaItems"];
+                                             NSUInteger index = [mutableArrayWithKVO indexOfObject:mediaItem];
+                                             [mutableArrayWithKVO replaceObjectAtIndex:index withObject:mediaItem];
+                                         }
+                                     } failure:^(AFHTTPRequestOperation *operation, NSError *error) {
+                                         NSLog(@"Error downloading image: %@", error);
+                                     }];
     }
 }
```

Run the app. Everything should work identically, but now it's running using a powerful networking library. This foundation will allow us to easily extend the capabilities of our app in future checkpoints.

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

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

Currently, if an image fails to download, it's never retried. You can simulate this by commenting out the line that decodes the image, and then relaunching the app:

BLCMedia.m

```
        self.mediaURL = [aDecoder decodeObjectForKey:NSStringFromSelector(@selector(mediaURL))];
-         self.image = [aDecoder decodeObjectForKey:NSStringFromSelector(@selector(image))];
+ //        self.image = [aDecoder decodeObjectForKey:NSStringFromSelector(@selector(image))];
        self.caption = [aDecoder decodeObjectForKey:NSStringFromSelector(@selector(caption))];
```

In the next checkpoint, we'll write a solution that allows the images to be automatically retried later if they fail for some reason.

You'll need to use a `UITapGestureRecognizer` with `numberOfTouchesRequired` set to `2`.

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 AFNetworking'
$ git checkout master
$ git merge install-afnetworking
$ git push
```

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

assignment completed

---

Send


hello@bloc.io

Considering enrolling? (404) 480-2562

Partnership / Corporate Inquiries? (650) 741-5682

Roadmap · Previous Checkpoint Jump to Submission · Next Checkpoir

# Retrying Image Downloads

> *"'Tis a lesson you should heed:*
> *Try, try, try again.*
> *If at first you don't succeed,*
> *Try, try, try again."*
>
> *- Thomas H. Palmer, 'Teacher's Manual'*

As discussed in the assignment section of the prior checkpoint, image downloads can sometimes fail. There are many reasons that your Internet connection may fail. Here are some examples:

- the user is on a train that went underground
- the user is in another country and has international roaming disabled
- the user's connection is slow, and the download timed out
- the user's cellular carrier doesn't support simultaneous voice and data, and the user received a phone call during download

In this checkpoint, we'll look at one solution for retrying image downloads after the initial attempt fails.

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

Terminal

```
$ git checkout -b retry-image-downloads
```

## Persisting Download State

We must track which media items have been downloaded, and which still need to be downloaded.

Let's start by declaring an enumeration in **`BLCMedia.h`**:

BLCMedia.h

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

+ typedef NS_ENUM(NSInteger, BLCMediaDownloadState) {
+     BLCMediaDownloadStateNeedsImage             = 0,
+     BLCMediaDownloadStateDownloadInProgress     = 1,
+     BLCMediaDownloadStateNonRecoverableError    = 2,
+     BLCMediaDownloadStateHasImage               = 3
+ };
+
@class BLCUser;

@interface BLCMedia : NSObject <NSCoding>
```

In case you don't remember **`typedef NS_ENUM`** from earlier, here's a quick refresher. This code declares **`BLCMediaDownloadState`** as equivaler to **`NSInteger`**, with four predefined values (**`0`**, **`1`**, **`2`**, and **`3`**.) A **`BLCMediaDownloadState`** can theoretically be used anywhere an **`NSInteger`** can,

```
NSInteger two = 2;
BLCMediaDownloadState downloadState = BLCMediaDownloadStateNonRecoverableError;
NSInteger four = two + downloadState;
/* four == 4, because BLCMediaDownloadStateNonRecoverableError == 2 */
```

We'll keep track of an individual media item's download state in a property:

BLCMedia.h

```
 @property (nonatomic, strong) NSURL *mediaURL;
 @property (nonatomic, strong) UIImage *image;
+ @property (nonatomic, assign) BLCMediaDownloadState downloadState;

 @property (nonatomic, strong) NSString *caption;
```

Note that this property is **`assign`** instead of **`strong`**. This is because **`BLCMediaDownloadState`** (aka **`NSInteger`**) is a simple type, not an object.

In **`init`**, let's set a default value:

BLCMedia.m

```
         if (standardResolutionImageURL) {
             self.mediaURL = standardResolutionImageURL;
+              self.downloadState = BLCMediaDownloadStateNeedsImage;
+          } else {
+              self.downloadState = BLCMediaDownloadStateNonRecoverableError;
         }

         NSDictionary *captionDictionary = mediaDictionary[@"caption"];
```

> *If we hadn't set a default value, the default value would automatically be set to* **`BLCMediaDownloadStateNeedsImage`***. This is because properties that are simple types are initialized to* **`0`** *by default (similar to how objects are* **`nil`** *by default).*

We'll also need to set **`self.downloadState`** in **`initWithCoder:`**:

BLCMedia.m

```
         self.user = [aDecoder decodeObjectForKey:NSStringFromSelector(@selector(user))];
         self.mediaURL = [aDecoder decodeObjectForKey:NSStringFromSelector(@selector(mediaURL))];
         self.image = [aDecoder decodeObjectForKey:NSStringFromSelector(@selector(image))];
+
+          if (self.image) {
+              self.downloadState = BLCMediaDownloadStateHasImage;
+          } else if (self.mediaURL) {
+              self.downloadState = BLCMediaDownloadStateNeedsImage;
+          } else {
+              self.downloadState = BLCMediaDownloadStateNonRecoverableError;
+          }
+
         self.caption = [aDecoder decodeObjectForKey:NSStringFromSelector(@selector(caption))];
         self.comments = [aDecoder decodeObjectForKey:NSStringFromSelector(@selector(comments))];
```

This way, after relaunching the app, saved media items will still have a download state.

Instead of encoding and decoding **`self.downloadState`** like we do with the other properties, we're determining the download state based on the presence of **`self.image`** and **`self.mediaURL`**. This way, all stored media items with missing images will be retried at least once the following time the app launches.

> *If you wanted, you could encode and decode* **`self.downloadState`** *instead of taking this approach. Note that since*

> `[NSCoder -decodeIntegerForKey:]`.

## Updating the Download State

Now we'll update **`BLCDataSource`** to update this property as appropriate.

BLCDataSource.m

```
 - (void) downloadImageForMediaItem:(BLCMedia *)mediaItem {
    if (mediaItem.mediaURL && !mediaItem.image) {
+         mediaItem.downloadState = BLCMediaDownloadStateDownloadInProgress;
+
        [self.instagramOperationManager GET:mediaItem.mediaURL.absoluteString
                                 parameters:nil
                                    success:^(AFHTTPRequestOperation *operation, id responseObject) {
                                        if ([responseObject isKindOfClass:[UIImage class]]) {
                                            mediaItem.image = responseObject;
+                                             mediaItem.downloadState = BLCMediaDownloadStateHasImage;
                                            NSMutableArray *mutableArrayWithKVO = [self mutableArrayValueForKey:@"mediaItems"];
                                            NSUInteger index = [mutableArrayWithKVO indexOfObject:mediaItem];
                                            [mutableArrayWithKVO replaceObjectAtIndex:index withObject:mediaItem];
+                                         } else {
+                                             mediaItem.downloadState = BLCMediaDownloadStateNonRecoverableError;
                                        }
                                    } failure:^(AFHTTPRequestOperation *operation, NSError *error) {
                                        NSLog(@"Error downloading image: %@", error);
+
+
+                                        mediaItem.downloadState = BLCMediaDownloadStateNonRecoverableError;
+
+                                        if ([error.domain isEqualToString:NSURLErrorDomain]) {
+                                            // A networking problem
+                                            if (error.code == NSURLErrorTimedOut ||
+                                                error.code == NSURLErrorCancelled ||
+                                                error.code == NSURLErrorCannotConnectToHost ||
+                                                error.code == NSURLErrorNetworkConnectionLost ||
+                                                error.code == NSURLErrorNotConnectedToInternet ||
+                                                error.code == kCFURLErrorInternationalRoamingOff ||
+                                                error.code == kCFURLErrorCallIsActive ||
+                                                error.code == kCFURLErrorDataNotAllowed ||
+                                                error.code == kCFURLErrorRequestBodyStreamExhausted) {
+
+                                                // It might work if we try again
+                                                mediaItem.downloadState = BLCMediaDownloadStateNeedsImage;
+                                            }
+                                        }
                                    }];
    }
 }
```

First, we set the download state to **`BLCMediaDownloadStateDownloadInProgress`** when we begin.

Once the operation is complete, if the image was set successfully, the state becomes **`BLCMediaDownloadStateHasImage`**.

If there's any error, we set the download state to **`BLCMediaDownloadStateNonRecoverableError`**. Finally, we check for a series of recoverable errors like timeouts and connectivity issues. If one of these errors occurs, we set the download state to **`BLCMediaDownloadStateNeedsImage`** instead. **NSHipster's `NSError` article** contains a list of common errors.

> ***Common Bug Warning.*** *When inspecting an* **`NSError`** *object, always check that* **`error.domain`** *is what you're expecting. For example, if there were a JSON parsing error instead of a URL loading error, you would get an error with the error domain* **`NSCocoaErrorDomain`** *instead of* **`NSURLErrorDomain`**.

## Using the Download State

Temporarily comment out the initial call to **`downloadImageForMediaItem:`**. You can use ⌘ **/** to comment out a line or group of lines:

BLCDataSource.m

```
        if (mediaItem) {
            [tmpMediaItems addObject:mediaItem];
-             [self downloadImageForMediaItem:mediaItem];
+ //            [self downloadImageForMediaItem:mediaItem];
        }
    }
```

We want other classes to be able to request that images be downloaded, so add **`downloadImageForMediaItem:`** to the public interface:

BLCDataSource.h

```
- (void) requestNewItemsWithCompletionHandler:(BLCNewItemCompletionBlock)completionHandler;
- (void) requestOldItemsWithCompletionHandler:(BLCNewItemCompletionBlock)completionHandler;

+ - (void) downloadImageForMediaItem:(BLCMedia *)mediaItem;
+
@end
```

Instead of downloading all the images as we get the media items, we'll check whether we need the images right before a cell displays. In the images table controller, implement **`tableView:willDisplayCell:forRowAtIndexPath:`**. According to the **UITableViewDelegate Protocol Reference**, a table view "sends this message to its delegate just before it uses **`cell`** to draw a row."

BLCImagesTableViewController.m

```
    return cell;
}

+ - (void) tableView:(UITableView *)tableView willDisplayCell:(UITableViewCell *)cell forRowAtIndexPath:(NSIndexPath *)indexPath {
+     BLCMedia *mediaItem = [BLCDataSource sharedInstance].mediaItems[indexPath.row];
+     if (mediaItem.downloadState == BLCMediaDownloadStateNeedsImage) {
+         [[BLCDataSource sharedInstance] downloadImageForMediaItem:mediaItem];
+     }
+ }
+
- (CGFloat) tableView:(UITableView *)tableView heightForRowAtIndexPath:(NSIndexPath *)indexPath {
    BLCMedia *item = [BLCDataSource sharedInstance].mediaItems[indexPath.row];
    return [BLCMediaTableViewCell heightForMediaItem:item width:CGRectGetWidth(self.view.frame)];
```

Run the app. Cells will initially appear without the image, and then the image will fade in.

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

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

Notice that the table may get really jumpy if you scroll through many cells really fast. Try to reduce the jumpiness.

Here are two possible solutions to experiment with:

1. Have a default value other than **`0`** for the image height constraint when an image isn't present.
2. Instead of loading images in **`tableView:willDisplayCell:forRowAtIndexPath:`**, you could only load images for the cells currently visible on the screen starting when the scrolling slows down. You can accomplish this by implementing the **`UIScrollViewDelegate`** methods **`scrollViewWillBeginDecelerating:`** and **`scrollViewDidScroll:`**. Read the documentation for **`[UIScrollView -`** **`decelerationRate]`**. You may also want to inspect **`dragging`** to avoid starting downloads while the user is still adjusting the scroll

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

Terminal

```
$ git add .
$ git commit -m 'Retry Image Downloads'
$ git checkout master
$ git merge retry-image-downloads
$ 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

# Like Button


What good is social media if you can't *like stuff*?

...and what good is liking stuff if you can't *unlike stuff*?

In this checkpoint, you'll make a heart-shaped button to like (or unlike) Instagram posts.

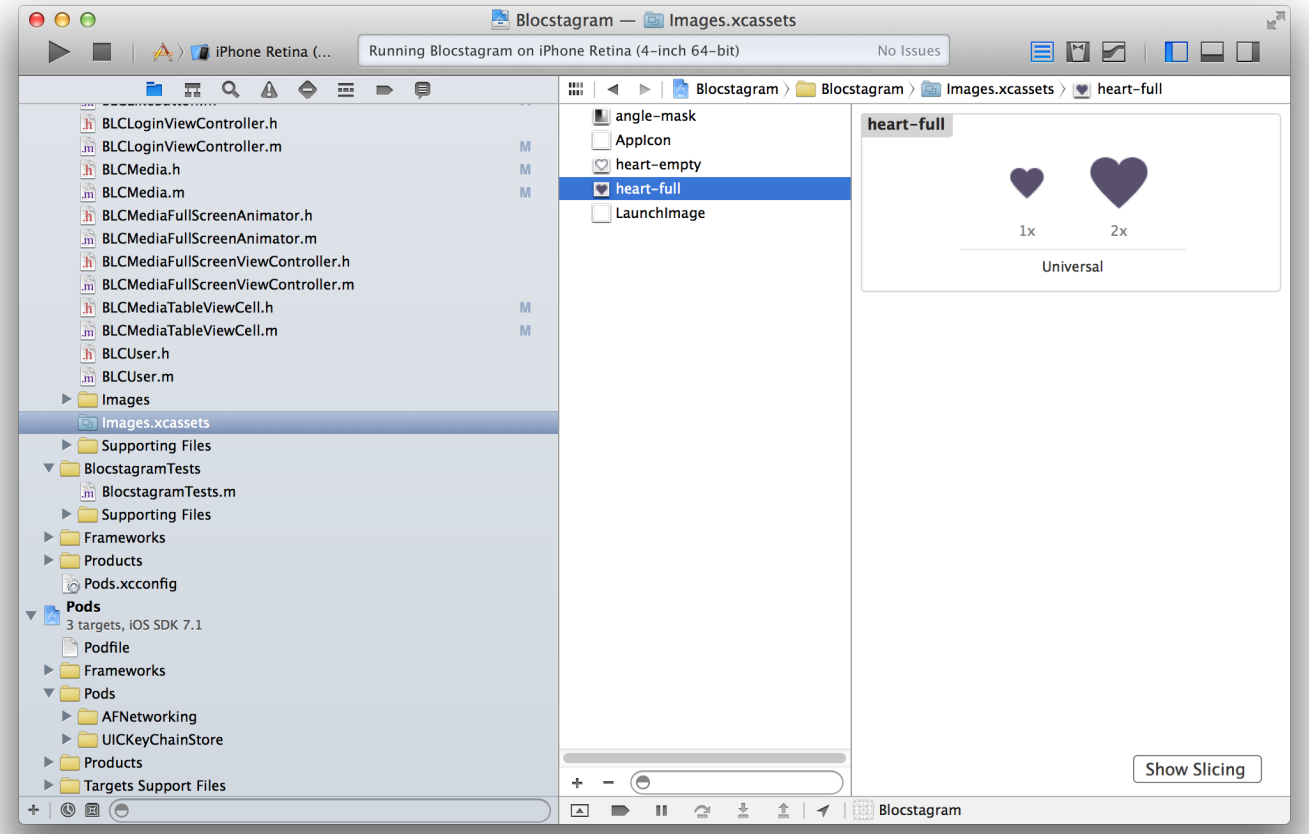
You'll learn a lot of new stuff along the way, including how to use Apple's Core Animation framework to draw an animated circle around the heart.

Since Core Animation is new to you, we'll start there.

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

Terminal

You'll need some images for this checkpoint. **Download `Like Button Resources.zip`**. Drag the 6 included images into the left column of your **`Images.xcassets`** file, so they appear like so:




**`.xcassets`** files are Xcode Asset Catalogs. They store multiple versions of the same image. In our case, we have two different images - one for devices with Retina Display, and one for devices with regular screens. You could additionally have separate iPad images. You can read more about them in Apple's **Asset Catalog documentation**.

# Introducing Core Animation

Core Animation renders graphics and creates animations. It's built in to iOS - every animation you see uses Core Animation under the hood.

There are whole books written on Core Animation, but here are a few basics so you understand what's going on:

- Visual content is represented by **layers**. These are represented by the **`CALayer`** class (or subclasses like **`CAShapeLayer`**). You already use these: **`UIView`** translates everything you do to **`CALayer`**s under the hood.
- Animations are made selecting before and after values for the **`CALayer`**s properties. For example, you can apply a rotation, opacity change, fade between colors, etc. These animations are represented by **`CAAnimation`** and its subclasses like **`CABasicAnimation`** and **`CAKeyframeAnimation`**.

It's easier to show by doing.

# Creating a Circular Spinner

Make a new **`UIView`** subclass called **`BLCCircleSpinnerView`**. Here's the **`.h`** file:

```
+ @interface BLCCircleSpinnerView : UIView
+
+ @property (nonatomic, assign) CGFloat strokeThickness;
+ @property (nonatomic, assign) CGFloat radius;
+ @property (nonatomic, strong) UIColor *strokeColor;
+
+ @end
```

## Building the Animation

In the **.m** file, add a property for a **CAShapeLayer**:

BLCCircleSpinnerView.m

```
#import "BLCCircleSpinnerView.h"

+ @interface BLCCircleSpinnerView ()
+
+ @property (nonatomic, strong) CAShapeLayer *circleLayer;
+
+ @end
```

We'll create **circleLayer** by overriding the getter, and creating it the first time it's called. (This is called **lazy instantiation**.)

There's a lot of code here, and we'll step through it bit by bit.

BLCCircleSpinnerView.m

```
+     if(!_circleLayer) {
+         CGPoint arcCenter = CGPointMake(self.radius+self.strokeThickness/2+5, self.radius+self.strokeThickness/2+5);
+         CGRect rect = CGRectMake(0, 0, arcCenter.x*2, arcCenter.y*2);
+
+         UIBezierPath* smoothedPath = [UIBezierPath bezierPathWithArcCenter:arcCenter
+                                                                     radius:self.radius
+                                                                 startAngle:M_PI*3/2
+                                                                   endAngle:M_PI/2+M_PI*5
+                                                                  clockwise:YES];
+
+         _circleLayer = [CAShapeLayer layer];
+         _circleLayer.contentsScale = [[UIScreen mainScreen] scale];
+         _circleLayer.frame = rect;
+         _circleLayer.fillColor = [UIColor clearColor].CGColor;
+         _circleLayer.strokeColor = self.strokeColor.CGColor;
+         _circleLayer.lineWidth = self.strokeThickness;
+         _circleLayer.lineCap = kCALineCapRound;
+         _circleLayer.lineJoin = kCALineJoinBevel;
+         _circleLayer.path = smoothedPath.CGPath;
+
+         CALayer *maskLayer = [CALayer layer];
+         maskLayer.contents = (id)[[UIImage imageNamed:@"angle-mask"] CGImage];
+         maskLayer.frame = _circleLayer.bounds;
+         _circleLayer.mask = maskLayer;
+
+         CFTimeInterval animationDuration = 1;
+         CAMediaTimingFunction *linearCurve = [CAMediaTimingFunction functionWithName:kCAMediaTimingFunctionLinear];
+
+         CABasicAnimation *animation = [CABasicAnimation animationWithKeyPath:@"transform.rotation"];
+         animation.fromValue = @0;
+         animation.toValue = @(M_PI*2);
+         animation.duration = animationDuration;
+         animation.timingFunction = linearCurve;
+         animation.removedOnCompletion = NO;
+         animation.repeatCount = INFINITY;
+         animation.fillMode = kCAFillModeForwards;
+         animation.autoreverses = NO;
+         [_circleLayer.mask addAnimation:animation forKey:@"rotate"];
+
+         CAAnimationGroup *animationGroup = [CAAnimationGroup animation];
+         animationGroup.duration = animationDuration;
+         animationGroup.repeatCount = INFINITY;
+         animationGroup.removedOnCompletion = NO;
+         animationGroup.timingFunction = linearCurve;
+
+         CABasicAnimation *strokeStartAnimation = [CABasicAnimation animationWithKeyPath:@"strokeStart"];
+         strokeStartAnimation.fromValue = @0.015;
+         strokeStartAnimation.toValue = @0.515;
+
+         CABasicAnimation *strokeEndAnimation = [CABasicAnimation animationWithKeyPath:@"strokeEnd"];
+         strokeEndAnimation.fromValue = @0.485;
+         strokeEndAnimation.toValue = @0.985;
+
+         animationGroup.animations = @[strokeStartAnimation, strokeEndAnimation];
+         [_circleLayer addAnimation:animationGroup forKey:@"progress"];
+
+     }
+     return _circleLayer;
+ }
```

Let's go through this a little bit at a time.

> *It will be difficult to understand how this all works only by reading the checkpoint and documentation. At the end of this checkpoint, spend some time playing with the different values and animation settings to get a better sense of what they do.*

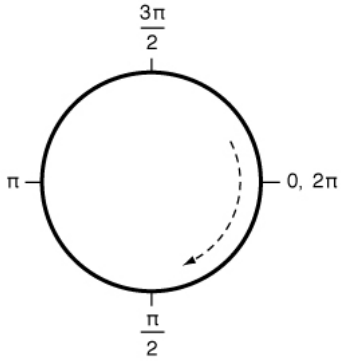
```
CGPoint arcCenter = CGPointMake(self.radius+self.strokeThickness/2+5, self.radius+self.strokeThickness/2+5);
```

The first line calculates a **`CGPoint`** representing the center of the arc. (In our case, the arc is an entire circle.)

Then, **`arcCenter`** is used to construct a **`CGRect`**. Our spinning circle will fit inside this rect.

```
UIBezierPath* smoothedPath = [UIBezierPath bezierPathWithArcCenter:arcCenter
                                                            radius:self.radius
                                                        startAngle:M_PI*3/2
                                                          endAngle:M_PI/2+M_PI*5
                                                         clockwise:YES];
```

We're now making a **`UIBezierPath`** object. (A bezier path is a path which can have both straight and curved line segments.) **`bezierPathWithArcCenter:radius:startAngle:endAngle:clockwise:`** makes a new bezier path in the shape of an arc, with the start and end angles in **radians**:




Basically, **`smoothedPath`** represents a smooth circular path.

```
_circleLayer = [CAShapeLayer layer];
_circleLayer.contentsScale = [[UIScreen mainScreen] scale];
_circleLayer.frame = rect;
_circleLayer.fillColor = [UIColor clearColor].CGColor;
_circleLayer.strokeColor = self.strokeColor.CGColor;
_circleLayer.lineWidth = self.strokeThickness;
_circleLayer.lineCap = kCALineCapRound;
_circleLayer.lineJoin = kCALineJoinBevel;
_circleLayer.path = smoothedPath.CGPath;
```

Here we're creating a **`CAShapeLayer`**. This is a core animation layer made from a bezier path. (Other layers can be made from other things, such as images.)

We set its **`contentScale`**, which is just like **`UIImage`**'s **`scale`** (**`1.0`** on regular screens; **`2.0`** on Retina Displays).

Its frame is set to **`rect`** from earlier. We want the center of the circle to be transparent (so we can see the heart), and the border to be the defined **`strokeColor`**. Note that core animation properties take **`CGColorRef`**s instead of **`UIColor`** objects, so we convert them using **the `CGColor` property**.

**`lineCap`** specifies the shape of the ends of the line. **`lineJoin`** specifies the shape of the joints between parts of the line.

Finally, we give the layer the path from our **`UIBezierPath`** object.

```
CALayer *maskLayer = [CALayer layer];
maskLayer.contents = (id)[[UIImage imageNamed:@"angle-mask"] CGImage];
maskLayer.frame = _circleLayer.bounds;
_circleLayer.mask = maskLayer;
```

We'll now make a mask layer and set its size to be same. A *mask layer* is an image or other layer that changes the opacity of its layer's content

Our image, **`angle-mask`** looks like this:

This will allow our circle to have a gradient on it.

Now we'll animate the mask in a circular motion.

```
CFTimeInterval animationDuration = 1;
CAMediaTimingFunction *linearCurve = [CAMediaTimingFunction functionWithName:kCAMediaTimingFunctionLinear];

CABasicAnimation *animation = [CABasicAnimation animationWithKeyPath:@"transform.rotation"];
animation.fromValue = @0;
animation.toValue = @(M_PI*2);
animation.duration = animationDuration;
animation.timingFunction = linearCurve;
animation.removedOnCompletion = NO;
animation.repeatCount = INFINITY;
animation.fillMode = kCAFillModeForwards;
animation.autoreverses = NO;
[_circleLayer.mask addAnimation:animation forKey:@"rotate"];
```

Here we set the animation duration to 1 second. (**`CFTimeInterval`**, like **`NSTimeInterval`**, is specified in seconds.)

We specify a linear animation (as opposed to easing in or out, for example.) This means the speed of the movement will stay the same throughout the entire animation.

We then specify that this animation will animate the layer's rotation transform from **`0`** to **`π * 2`** (one full circular turn.)

This animation will be repeated an infinite number of times. **`fillMode`** specifies what happens when the animation is complete (you can opt t hide layers once an animation has ended.) In our case, we specify **`kCAFillModeForwards`** to leave the layer on screen after the animation.

Finally, we add the animation to the layer.

Now that we've animated the mask, all that's left is to animate the line that draws the circle itself.

```
CAAnimationGroup *animationGroup = [CAAnimationGroup animation];
animationGroup.duration = animationDuration;
animationGroup.repeatCount = INFINITY;
animationGroup.removedOnCompletion = NO;
animationGroup.timingFunction = linearCurve;

CABasicAnimation *strokeStartAnimation = [CABasicAnimation animationWithKeyPath:@"strokeStart"];
strokeStartAnimation.fromValue = @0.015;
strokeStartAnimation.toValue = @0.515;

CABasicAnimation *strokeEndAnimation = [CABasicAnimation animationWithKeyPath:@"strokeEnd"];
strokeEndAnimation.fromValue = @0.485;
strokeEndAnimation.toValue = @0.985;

animationGroup.animations = @[strokeStartAnimation, strokeEndAnimation];
[_circleLayer addAnimation:animationGroup forKey:@"progress"];
```

Here, we're creating two **`CABasicAnimation`**s. One animates the start of the stroke; the other animates the end. Both animations are added t a **`CAAnimationGroup`**, which allows multiple animations to be grouped and run concurrently.

We add the animations to the circle layer, and we're done building the animation.

## Configuring the View

Let's create a method to ensure that the circle animation is positioned properly. We'll call it from a few places.

```
+ - (void)layoutAnimatedLayer {
+     [self.layer addSublayer:self.circleLayer];
+
+     self.circleLayer.position = CGPointMake(CGRectGetMidX(self.bounds), CGRectGetMidY(self.bounds));
+ }
```

This code just positions the circle layer in the center of the view.

When we add a subview to another view using **`[UIView -addSubview:]`**, the subview can react to this in **`[UIView -willMoveToSuperview:]`**. Let's implement this method to ensure our positioning is accurate:

BLCCircleSpinnerView.m

```
+ - (void)willMoveToSuperview:(UIView *)newSuperview {
+     if (newSuperview != nil) {
+         [self layoutAnimatedLayer];
+     }
+     else {
+         [self.circleLayer removeFromSuperlayer];
+         self.circleLayer = nil;
+     }
+ }
```

We'll also need to update the position of the layer if the frame changes:

BLCCircleSpinnerView.m

```
+ - (void)setFrame:(CGRect)frame {
+     [super setFrame:frame];
+
+     if (self.superview != nil) {
+         [self layoutAnimatedLayer];
+     }
+ }
```

If we change the radius of the circle, that will affect positioning as well. We can update this by recreating the circle layer any time the radius changes:

BLCCircleSpinnerView.m

```
+ - (void)setRadius:(CGFloat)radius {
+     _radius = radius;
+
+     [_circleLayer removeFromSuperlayer];
+     _circleLayer = nil;
+
+     [self layoutAnimatedLayer];
+ }
```

We should also inform **`self.circleLayer`** if the other two properties change (stroke width or color):

BLCCircleSpinnerView.m

```
+ - (void)setStrokeColor:(UIColor *)strokeColor {
+     _strokeColor = strokeColor;
+     _circleLayer.strokeColor = strokeColor.CGColor;
+ }
+
+ - (void)setStrokeThickness:(CGFloat)strokeThickness {
+     _strokeThickness = strokeThickness;
+     _circleLayer.lineWidth = _strokeThickness;
+ }
```

BLCCircleSpinnerView.m

```
+ - (id)initWithFrame:(CGRect)frame
+ {
+     self = [super initWithFrame:frame];
+     if (self) {
+         self.strokeThickness = 1;
+         self.radius = 12;
+         self.strokeColor = [UIColor purpleColor];
+     }
+     return self;
+ }
+
+ - (CGSize)sizeThatFits:(CGSize)size {
+     return CGSizeMake((self.radius+self.strokeThickness/2+5)*2, (self.radius+self.strokeThickness/2+5)*2);
+ }
```

This completes our circular spinning view. Now let's create a Like button that uses it.

# Creating a custom **UIButton**

Create a new class called **BLCLikeButton**, which will be a subclass of **UIButton**.

The **.h** file will simply define four possible states the button might be in, and expose a property for storing this state:

BLCLikeButton.h

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

+ typedef NS_ENUM(NSInteger, BLCLikeState) {
+     BLCLikeStateNotLiked            = 0,
+     BLCLikeStateLiking              = 1,
+     BLCLikeStateLiked               = 2,
+     BLCLikeStateUnliking            = 3
+ };
+
@interface BLCLikeButton : UIButton
+
+ /**
+  The current state of the like button. Setting to BLCLikeButtonNotLiked or BLCLikeButtonLiked will display an empty heart or a hear
+  */
+ @property (nonatomic, assign) BLCLikeState likeButtonState;

@end
```

In the **.m** file, import the spinner view class, define the image names, and make a property for storing the spinner view:

BLCLikeButton.m

```
#import "BLCLikeButton.h"
+ #import "BLCCircleSpinnerView.h"
+
+ #define kLikedStateImage @"heart-full"
+ #define kUnlikedStateImage @"heart-empty"
+
+ @interface BLCLikeButton ()
+
+ @property (nonatomic, strong) BLCCircleSpinnerView *spinnerView;
+
+ @end
```

In the initializer, we'll create the spinner view, and set up the button.

```
@implementation BLCLikeButton

+ - (instancetype) init {
+     self = [super init];
+
+     if (self) {
+         self.spinnerView = [[BLCCircleSpinnerView alloc] initWithFrame:CGRectMake(0, 0, 44, 44)];
+         [self addSubview:self.spinnerView];
+
+         self.imageView.contentMode = UIViewContentModeScaleAspectFit;
+
+         self.contentEdgeInsets = UIEdgeInsetsMake(10, 10, 10, 10);
+         self.contentVerticalAlignment = UIControlContentVerticalAlignmentTop;
+
+         self.likeButtonState = BLCLikeStateNotLiked;
+     }
+
+     return self;
+ }
```

There's two new properties here:

- `contentEdgeInsets` provides a buffer between the edge of the button and the content.
- `contentVerticalAlignment` specifies the alignment of the button's content. By default it's centered, but we want it at the top so that th like button isn't misaligned on photos with longer captions.

We also set the default state to "not liked".

The spinner view's frame should be updated whenever the button's frame changes:

BLCLikeButton.m

```
+ - (void) layoutSubviews {
+     [super layoutSubviews];
+     self.spinnerView.frame = self.imageView.frame;
+ }
```

Finally, we need to write code that updates the button based on the set state:

BLCLikeButton.m

```
+      _likeButtonState = likeState;
+
+      NSString *imageName;
+
+      switch (_likeButtonState) {
+          case BLCLikeStateLiked:
+          case BLCLikeStateUnliking:
+              imageName = kLikedStateImage;
+              break;
+
+          case BLCLikeStateNotLiked:
+          case BLCLikeStateLiking:
+              imageName = kUnlikedStateImage;
+      }
+
+      switch (_likeButtonState) {
+          case BLCLikeStateLiking:
+          case BLCLikeStateUnliking:
+              self.spinnerView.hidden = NO;
+              self.userInteractionEnabled = NO;
+              break;
+
+          case BLCLikeStateLiked:
+          case BLCLikeStateNotLiked:
+              self.spinnerView.hidden = YES;
+              self.userInteractionEnabled = YES;
+      }
+
+
+      [self setImage:[UIImage imageNamed:imageName] forState:UIControlStateNormal];
+  }

  @end
```

This setter updates the button's image and **`userInteractionEnabled`** property depending on the **`BLCLikeState`** passed in. It also shows or hides the spinner view as appropriate.

# Updating the Media Model to Track Like State

The button is just a *view* - it just shows what it's told. In order to display accurate information, we'll need to know whether the user has liked a image or not.

In **`BLCMedia.h`**, import the like button (so that **`BLCLikeState`**, the list of possible states, is defined):

BLCMedia.h

```
 #import <Foundation/Foundation.h>
+ #import "BLCLikeButton.h"

 typedef NS_ENUM(NSInteger, BLCMediaDownloadState) {
     BLCMediaDownloadStateNeedsImage             = 0,
```

Add a property to keep the media item's state:

BLCMedia.h

```
 @property (nonatomic, strong) NSArray *comments;

+ @property (nonatomic, assign) BLCLikeState likeState;
+
 - (instancetype) initWithDictionary:(NSDictionary *)mediaDictionary;
```

In **`initWithDictionary:`**, figure out whether the user has already liked the image:

```
        }

        self.comments = commentsArray;
+
+         BOOL userHasLiked = [mediaDictionary[@"user_has_liked"] boolValue];
+
+         self.likeState = userHasLiked ? BLCLikeStateLiked : BLCLikeStateNotLiked;
    }

    return self;
```

Add the appropriate **encode…** and **decode…** calls:

BLCMedia.m

```
        self.caption = [aDecoder decodeObjectForKey:NSStringFromSelector(@selector(caption))];
        self.comments = [aDecoder decodeObjectForKey:NSStringFromSelector(@selector(comments))];
+         self.likeState = [aDecoder decodeIntegerForKey:NSStringFromSelector(@selector(likeState))];
    }
```

BLCMedia.m

```
    [aCoder encodeObject:self.caption forKey:NSStringFromSelector(@selector(caption))];
    [aCoder encodeObject:self.comments forKey:NSStringFromSelector(@selector(comments))];
+     [aCoder encodeInteger:self.likeState forKey:NSStringFromSelector(@selector(likeState))];
}
```

Now that media items know whether they've been liked, let's add the button to the table cell.

# Adding the Like Button to the Table Cell

In the delegate protocol, add a method indicating that the button was pressed:

BLCMediaTableViewCell.h

```
- (void) cell:(BLCMediaTableViewCell *)cell didTapImageView:(UIImageView *)imageView;
- (void) cell:(BLCMediaTableViewCell *)cell didLongPressImageView:(UIImageView *)imageView;
+ - (void) cellDidPressLikeButton:(BLCMediaTableViewCell *)cell;

@end
```

In the **.m** file, import the like button:

BLCMediaTableViewCell.m

```
#import "BLCComment.h"
#import "BLCUser.h"
+ #import "BLCLikeButton.h"
```

Create a property for it:

BLCMediaTableViewCell.m

```
@property (nonatomic, strong) UITapGestureRecognizer *tapGestureRecognizer;
@property (nonatomic, strong) UILongPressGestureRecognizer *longPressGestureRecognizer;

+ @property (nonatomic, strong) BLCLikeButton *likeButton;
+
@end
```

1. Create the button
2. Add it to the view hierarchy
3. Update the layout constraints with its location

BLCMediaTableViewCell.m

```
        self.commentLabel.numberOfLines = 0;
        self.commentLabel.backgroundColor = commentLabelGray;

+         self.likeButton = [[BLCLikeButton alloc] init];
+         [self.likeButton addTarget:self action:@selector(likePressed:) forControlEvents:UIControlEventTouchUpInside];
+         self.likeButton.backgroundColor = usernameLabelGray;
+
+         for (UIView *view in @[self.mediaImageView, self.usernameAndCaptionLabel, self.commentLabel, self.likeButton]) {
-         for (UIView *view in @[self.mediaImageView, self.usernameAndCaptionLabel, self.commentLabel]) {
            [self.contentView addSubview:view];
            view.translatesAutoresizingMaskIntoConstraints = NO;
        }

-         NSDictionary *viewDictionary = NSDictionaryOfVariableBindings(_mediaImageView, _usernameAndCaptionLabel, _commentLabel);
+         NSDictionary *viewDictionary = NSDictionaryOfVariableBindings(_mediaImageView, _usernameAndCaptionLabel, _commentLabel, _li

        [self.contentView addConstraints:[NSLayoutConstraint constraintsWithVisualFormat:@"H:|[_mediaImageView]|" options:kNilOptions
-         [self.contentView addConstraints:[NSLayoutConstraint constraintsWithVisualFormat:@"H:|[_usernameAndCaptionLabel]|" options:
+         [self.contentView addConstraints:[NSLayoutConstraint constraintsWithVisualFormat:@"H:|[_usernameAndCaptionLabel][_likeButto
        [self.contentView addConstraints:[NSLayoutConstraint constraintsWithVisualFormat:@"H:|[_commentLabel]|" options:kNilOptions m

        [self.contentView addConstraints:[NSLayoutConstraint constraintsWithVisualFormat:@"V:|[_mediaImageView][_usernameAndCaptionLa
```

In the auto-layout constraints, we're setting an explicit width of **38**.

In **setMediaItem:** we need to display the correct state on the button:

BLCMediaTableViewCell.m

```
    self.usernameAndCaptionLabel.attributedText = [self usernameAndCaptionString];
    self.commentLabel.attributedText = [self commentString];
+     self.likeButton.likeButtonState = mediaItem.likeState;
}
```

When the button is tapped, we inform the delegate:

BLCMediaTableViewCell.m

```
+ #pragma mark - Liking
+
+ - (void) likePressed:(UIButton *)sender {
+     [self.delegate cellDidPressLikeButton:self];
+ }
+
#pragma mark - Image View
```

We need to update the data source to tell the Instagram API when a user likes or unlikes an image.

# Updating the Data Source

Let's add a new method called **toggleLikeOnMediaItem:** to the data source.

In the **.h** file:

BLCDataSource.h

```
+ - (void) toggleLikeOnMediaItem:(BLCMedia *)mediaItem;
+
@end
```

To understand the implementation, we'll take a quick detour and talk about HTTP verbs.

## HTTP Verbs

> *HTTP verbs are sometimes called "HTTP methods".*

When we connect to a server, we make a **`NSURLRequest`** object. As part of the object, we need to specify the action we want the server to take

The most common verb - the one we've used in this app so far - is **`GET`**.

When you **`GET`** a URL, you are asking for the most recent version of some resource. Responses to **`GET`** requests never create, modify, or remove data, they only transmit it.

Here's a list of common HTTP verbs:

| Verb | What it does |
|---|---|
| **`GET`** | Requests a representation of the resource at the specified URL. |
| **`HEAD`** | Requests only the HTTP headers for the response identical to the one that would correspond to a **`GET`** request. |
| **`POST`** | Creates a new resource related to the one at the specified URL. |
| **`PUT`** | Update the resource at the specified URL (or create it if it's not there). |
| **`DELETE`** | Deletes the resource at the specified URL. |

A "resource" can be anything - a like, a comment, an image, a caption, etc.

According to the **Instagram API documentation on likes**, when we like an image, we'll create a **`POST`** request. When we unlike an image, we'll create a **`DELETE`** request.

Here's how the implementation looks, along with a helper method that will reload the row:

BLCDataSource.m

```
+
+ - (void) toggleLikeOnMediaItem:(BLCMedia *)mediaItem {
+     NSString *urlString = [NSString stringWithFormat:@"media/%@/likes", mediaItem.idNumber];
+     NSDictionary *parameters = @{@"access_token": self.accessToken};
+
+     if (mediaItem.likeState == BLCLikeStateNotLiked) {
+
+         mediaItem.likeState = BLCLikeStateLiking;
+
+         [self.instagramOperationManager POST:urlString parameters:parameters success:^(AFHTTPRequestOperation *operation, id respon
+             mediaItem.likeState = BLCLikeStateLiked;
+             [self reloadMediaItem:mediaItem];
+         } failure:^(AFHTTPRequestOperation *operation, NSError *error) {
+             mediaItem.likeState = BLCLikeStateNotLiked;
+             [self reloadMediaItem:mediaItem];
+         }];
+
+     } else if (mediaItem.likeState == BLCLikeStateLiked) {
+
+         mediaItem.likeState = BLCLikeStateUnliking;
+
+         [self.instagramOperationManager DELETE:urlString parameters:parameters success:^(AFHTTPRequestOperation *operation, id resp
+             mediaItem.likeState = BLCLikeStateNotLiked;
+             [self reloadMediaItem:mediaItem];
+         } failure:^(AFHTTPRequestOperation *operation, NSError *error) {
+             mediaItem.likeState = BLCLikeStateLiked;
+             [self reloadMediaItem:mediaItem];
+         }];
+
+     }
+
+     [self reloadMediaItem:mediaItem];
+ }
+
+ - (void) reloadMediaItem:(BLCMedia *)mediaItem {
+     NSMutableArray *mutableArrayWithKVO = [self mutableArrayValueForKey:@"mediaItems"];
+     NSUInteger index = [mutableArrayWithKVO indexOfObject:mediaItem];
+     [mutableArrayWithKVO replaceObjectAtIndex:index withObject:mediaItem];
+ }
```

Let's walk through this code. Assume **`mediaItem.likeState`** is **`BLCLikeStateNotLiked`**. Therefore, when the user calls this method, they are attempting to like the image. We set **`mediaItem.likeState`** to **`BLCLikeStateLiking`** (which will show the circular animation.) We then create a **`POST`** request to the URL generated at the top of the method (**`urlString`**). If it worked, then we update the state to **`BLCLikeStateLiked`**; if it didn't, we revert the state back to **`BLCLikeStateNotLiked`**. Either way, we reload the row again to update the comment.

Similar logic is applied when unliking an image, with the HTTP verb switched to **`DELETE`**.

# Connecting the Cell to the Data Source

The last thing to do is to connect the cell to the new data source method we added. This is simple enough - open up the images table controller:

BLCImagesTableViewController.m

```
     }
 }

+ - (void) cellDidPressLikeButton:(BLCMediaTableViewCell *)cell {
+     [[BLCDataSource sharedInstance] toggleLikeOnMediaItem:cell.mediaItem];
+ }
+
 #pragma mark - UIViewControllerTransitioningDelegate

 - (id<UIViewControllerAnimatedTransitioning>)animationControllerForPresentedController:(UIViewController *)presented
```

## Test it out

Run the app, and try tapping the heart button.

When we tap the heart button, it spins momentarily, and then stops. Looks like there was some sort of problem.

We can hypothesize from the fact that the button spins for a second that there's some sort of issue with the response to the API request.

Let's look at our API code:

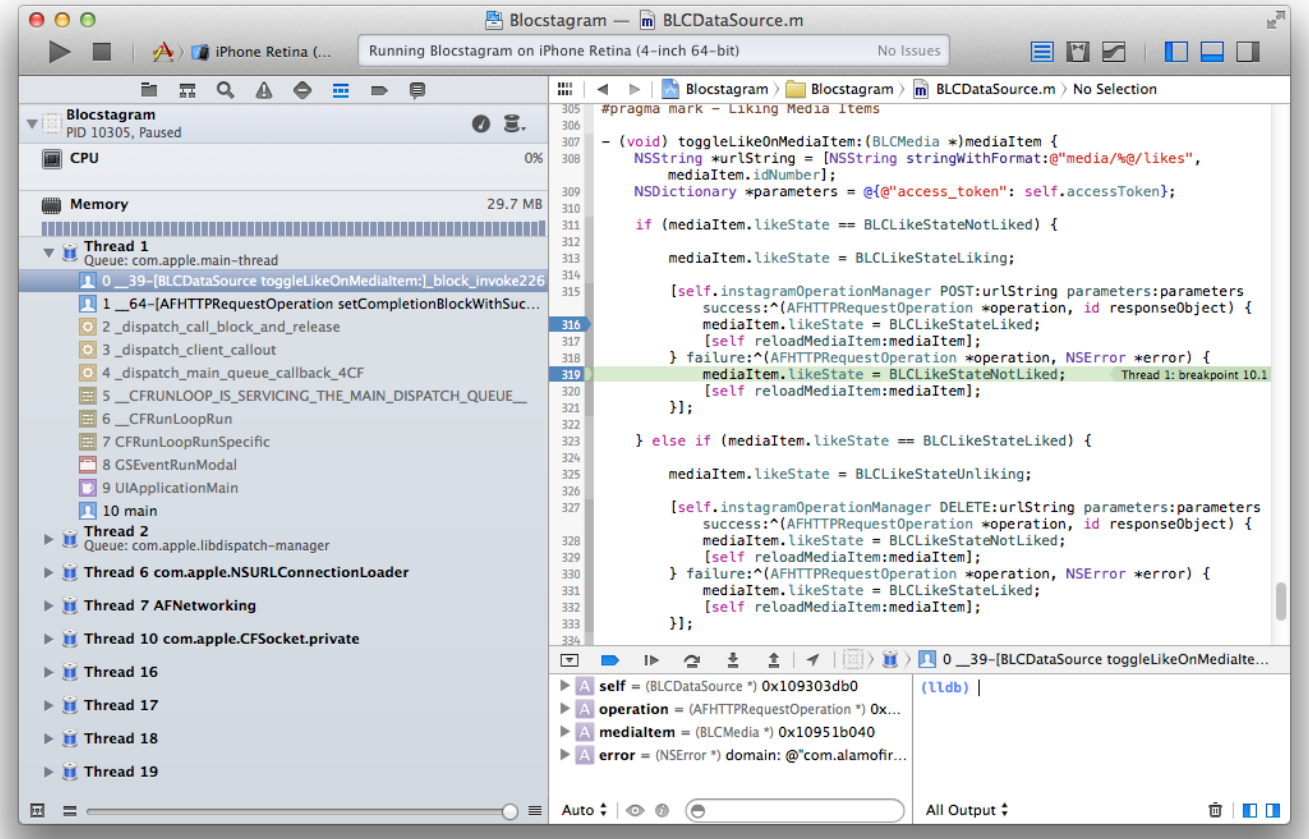
```
if (mediaItem.likeState == BLCLikeStateNotLiked) {

    mediaItem.likeState = BLCLikeStateLiking;

    [self.instagramOperationManager POST:urlString parameters:parameters success:^(AFHTTPRequestOperation *operation, id responseO
        mediaItem.likeState = BLCLikeStateLiked;
        [self reloadMediaItem:mediaItem];
    } failure:^(AFHTTPRequestOperation *operation, NSError *error) {
        mediaItem.likeState = BLCLikeStateNotLiked;
        [self reloadMediaItem:mediaItem];
    }];
```

Let's set two breakpoints, at the beginning of the success and failure blocks. So add breakpoints on these two lines:

1. **`mediaItem.likeState = BLCLikeStateLiked;`**
2. **`mediaItem.likeState = BLCLikeStateNotLiked;`**

You can add or remove a breakpoint using ⌘ ****.

Now that the breakpoints are on, tap the heart. The breakpoint in the failure block should trigger:



At the **(lldb)** prompt, type **po error** to print the description of the **NSError** object:

Debug

```
(lldb) po error
Error Domain=com.alamofire.error.serialization.response Code=-1011 "Request failed: bad request (400)" UserInfo=0x10932aeb0 {com.alamo
    "Cache-Control" = "private, no-cache, no-store, must-revalidate";
    Connection = "keep-alive";
    "Content-Language" = en;
    "Content-Type" = "application/json; charset=utf-8";
    Date = "Tue, 22 Jul 2014 00:04:00 GMT";
    Expires = "Sat, 01 Jan 2000 00:00:00 GMT";
    Pragma = "no-cache";
    Server = nginx;
    "Set-Cookie" = "csrftoken=5711f598307535f1b1716fce2ada846a; expires=Tue, 21-Jul-2015 00:04:00 GMT; Max-Age=31449600; Path=/";
    "Transfer-Encoding" = Identity;
    Vary = "Cookie, Accept-Language";
    "X-Ratelimit-Limit" = 5000;
    "X-Ratelimit-Remaining" = 4998;
} }, NSLocalizedDescription=Request failed: bad request (400), NSErrorFailingURLKey=https://api.instagram.com/v1/media/769549834046541
```

The server returned an **HTTP 400** error. Let's see if any additional information is available in the response object. Type **po operation.responseObject**.

Debug

```
(lldb) po operation.responseObject
{
    meta =     {
        code = 400;
        "error_message" = "This request requires scope=likes, but this access token is not authorized with this scope. The user must r
        "error_type" = OAuthPermissionsException;
    };
}
```

Let's take a look at the **Instagram API authentication documentation**:

> *Currently, all apps have basic read access by default. If all you want to do is access data then you do not need to specify a scope (the "basic" scope will be granted automatically).*
>
> *However, if you plan on asking for extended access such as liking, commenting, or managing friendships, you'll have to specify these scopes in your authorization request. Here are the scopes we currently support:*
>
> - ***basic*** *- to read any and all data related to a user (e.g. following/followed-by lists, photos, etc.) (granted by default)*
> - ***comments*** *- to create or delete comments on a user's behalf*
> - ***relationships*** *- to follow and unfollow users on a user's behalf*
> - ***likes*** *- to like and unlike items on a user's behalf*
>
> *If you'd like to request multiple scopes at once, simply separate the scopes by a space. In the url, this equates to an escaped space ("+"). So if you're requesting the likes and comments permission, the parameter will look like this:*
>
> ```
> scope=likes+comments
> ```

Let's update the login controller to request these permissions:

BLCLoginViewController.m

```
    // Do any additional setup after loading the view.

-     NSString *urlString = [NSString stringWithFormat:@"https://instagram.com/oauth/authorize/?client_id=%@&redirect_uri=%@&response
+     NSString *urlString = [NSString stringWithFormat:@"https://instagram.com/oauth/authorize/?client_id=%@&scope=likes+comments+rel
    NSURL *url = [NSURL URLWithString:urlString];
```

Since we don't have a logout function yet, reset the simulator to force us to login again:

1. Switch to the **iOS Simulator** app.
2. In the **iOS Simulator** menu, select **Reset Content and Settings...**.
3. In the alert view, press **Reset**.

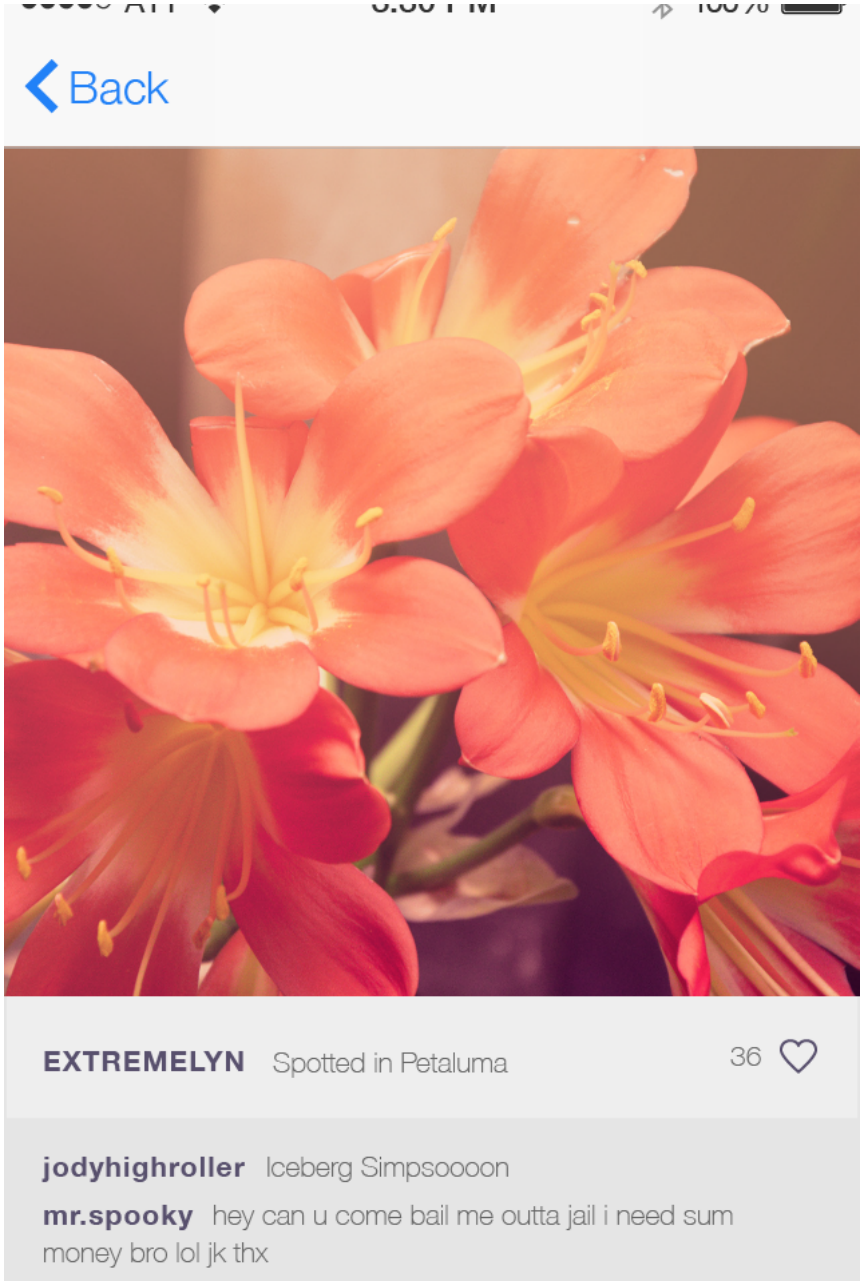
Now disable breakpoints, launch the app again, and login.

You can now like and unlike photos!

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

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

1. Play around with the animation and shape values in `BLCCircleSpinnerView`.
2. Add a label to `BLCMediaTableViewCell` that displays the current number of likes like this:




Make sure it updates when the user likes and unlikes an image.

3. Currently, the like state isn't saved to disk after a user (un)likes an image. Save the change to disk when the state successfully changes.

**Optional Extra Credit Assignment:** When the like state changes, we reload the entire cell to show the updated state. This shows a flickering effect. Instead, update only the state of the button without reloading the entire cell. Remember that due to table view cell reuse, the button that you need to update might be a different instance than the button that was tapped.

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 Like Button'
$ git checkout master
$ git merge like-button
$ git push
```

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

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

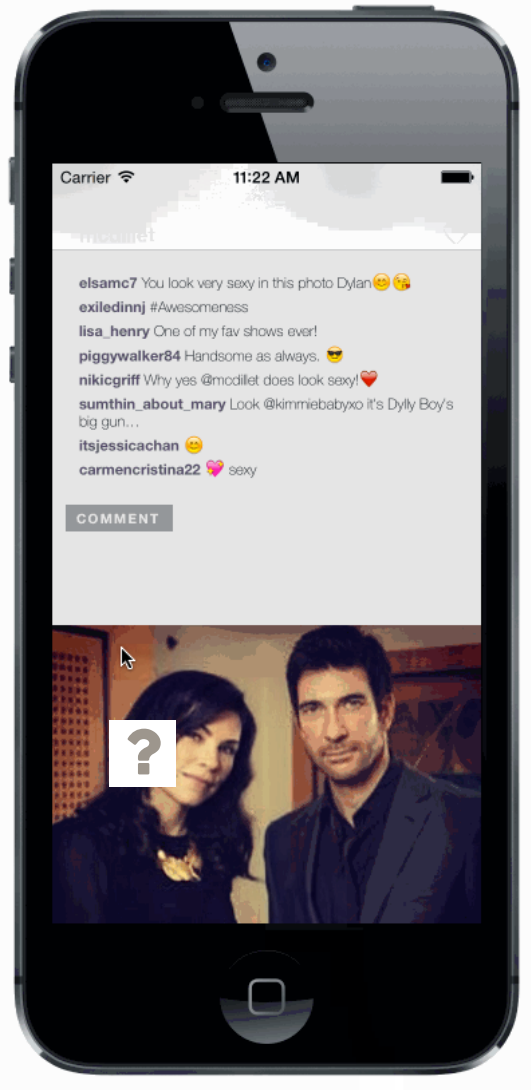
SIGN UP FOR OUR MAILING LIST

Send


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

Roadmap · Previous Checkpoint Jump to Submission · Next Checkpoir

# Comment View




> *'No comment' is a splendid expression. I am using it again and again.*
>
> *- Winston Churchill*

In this checkpoint, we'll combine much of what we've learned to create a comment box.

> ***NOTE:** Instagram has disabled the creation of comments from third party apps. This checkpoint exists to show you the code necessary to accomplish commenting, were you to obtain permission from Instagram. If you want your app to be able to post comments, you'll need to **request access to the `comments` endpoint**. (Anecdotally, it seems that most requests go unanswered.)*

- The delegate pattern
- Subclassing **`UIView`**
- **`NSNotification`**
- Model/View/Controller
- **`UIView`** animations

You'll also learn a bit about keyboard handling.

# Creating a Comment View

We're going to create a new view. Let's try to keep it decoupled from our model. In order to accomplish this, we'll make sure it has no idea what a **`BLCMedia`** or **`BLCComment`** is.

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

Terminal

```
$ git checkout -b comment-view
```

Let's start, as usual, by creating its interface. Create a new **`UIView`** subclass called **`BLCComposeCommentView`**. Here's the interface:

BLCComposeCommentView.h

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

+ @class BLCComposeCommentView;
+
+ @protocol BLCComposeCommentViewDelegate <NSObject>
+
+ - (void) commentViewDidPressCommentButton:(BLCComposeCommentView *)sender;
+ - (void) commentView:(BLCComposeCommentView *)sender textDidChange:(NSString *)text;
+ - (void) commentViewWillStartEditing:(BLCComposeCommentView *)sender;
+
+ @end
+
@interface BLCComposeCommentView : UIView

+ @property (nonatomic, weak) NSObject <BLCComposeCommentViewDelegate> *delegate;
+
+ @property (nonatomic, assign) BOOL isWritingComment;
+
+ @property (nonatomic, strong) NSString *text;
+
+ - (void) stopComposingComment;
+
@end
```

Its delegate protocol will inform its delegate when the user starts editing, updates the text, and presses the comment button.

We also have a **`BOOL`** called **`isWritingComment`** which determines if the user is currently editing a comment.

**`text`** contains the text of the comment, and will allow an external controller to set text.

A controller can send this view **`stopComposingComment`** if some external event means that the compose workflow should end and the keyboard should be dismissed.

Let's look at the implementation now.

Our view will be composed of a text view and a button, so let's add properties to store those, and declare that we conform to the **`UITextViewDelegate`** protocol:

```
+ @interface BLCComposeCommentView () <UITextViewDelegate>
+
+ @property (nonatomic, strong) UITextView *textView;
+ @property (nonatomic, strong) UIButton *button;
+
+ @end
```

In the initializer, we'll create and configure these objects and add them to the view hierarchy. We also create a new method to generate an attributed string for the button text:

BLCComposeCommentView.m

```
+ - (id)initWithFrame:(CGRect)frame
+ {
+     self = [super initWithFrame:frame];
+     if (self) {
+         // Initialization code
+         self.textView = [UITextView new];
+         self.textView.delegate = self;
+
+         self.button = [UIButton buttonWithType:UIButtonTypeCustom];
+         [self.button setAttributedTitle:[self commentAttributedString] forState:UIControlStateNormal];
+         [self.button addTarget:self action:@selector(commentButtonPressed:) forControlEvents:UIControlEventTouchUpInside];
+
+         [self addSubview:self.textView];
+         [self.textView addSubview:self.button];
+     }
+     return self;
+ }
+
+ - (NSAttributedString *) commentAttributedString {
+     NSString *baseString = NSLocalizedString(@"COMMENT", @"comment button text");
+     NSRange range = [baseString rangeOfString:baseString];
+
+     NSMutableAttributedString *commentString = [[NSMutableAttributedString alloc] initWithString:baseString];
+
+     [commentString addAttribute:NSFontAttributeName value:[UIFont fontWithName:@"HelveticaNeue-Bold" size:10] 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;
+ }
```

Note that **`self.button`** is a subview of **`self.textView`**, not **`self`**. This will be helpful when we want to wrap long comment text around a button.

Aside from that, all of this initializer should look familiar - we're creating both objects, setting their delegates and targets, and adding them.

There's one new thing inside **`commentAttributedString`**: the **`NSKernAttributeName`**. This number “specifies the number of points by which tc adjust kern-pair characters”. More simply, it increases character spacing.

Now we need to layout the views so that it looks correct. As you can see in the screenshot, we have two different states. While the user is scrolling, the comment button is gray on the left. Once the user starts editing the button slides to the right and fades to purple.

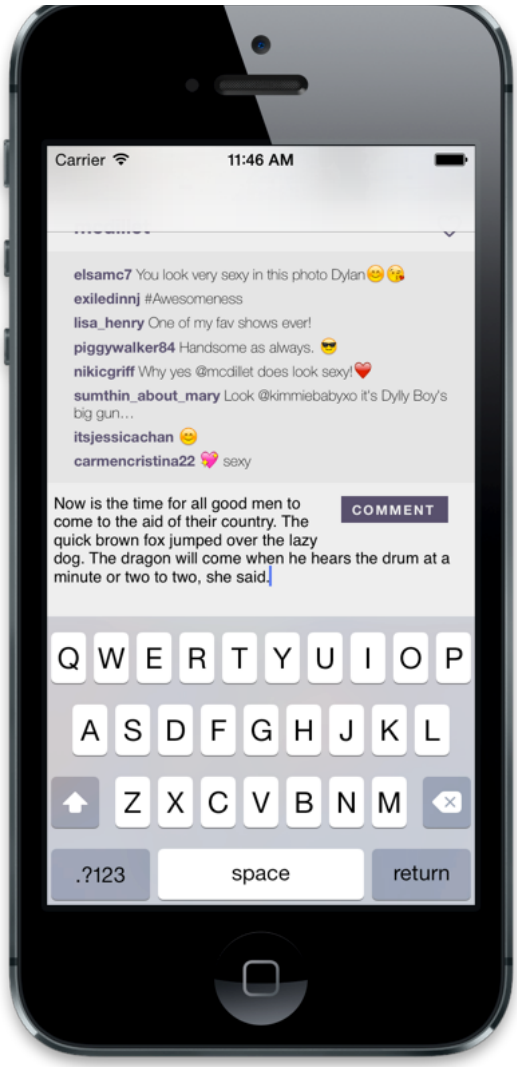
We'll define this programmatically in **`layoutSubviews`**:

BLCComposeCommentView.m

```
+     [super layoutSubviews];
+
+     self.textView.frame = self.bounds;
+
+     if (self.isWritingComment) {
+         self.textView.backgroundColor = [UIColor colorWithRed:0.933 green:0.933 blue:0.933 alpha:1]; /*#eeeeee*/
+         self.button.backgroundColor = [UIColor colorWithRed:0.345 green:0.318 blue:0.424 alpha:1]; /*#58516c*/
+
+         CGFloat buttonX = CGRectGetWidth(self.bounds) - CGRectGetWidth(self.button.frame) - 20;
+         self.button.frame = CGRectMake(buttonX, 10, 80, 20);
+     } else {
+         self.textView.backgroundColor = [UIColor colorWithRed:0.898 green:0.898 blue:0.898 alpha:1]; /*#e5e5e5*/
+         self.button.backgroundColor = [UIColor colorWithRed:0.6 green:0.6 blue:0.6 alpha:1]; /*#999999*/
+
+         self.button.frame = CGRectMake(10, 10, 80, 20);
+     }
+
+     CGSize buttonSize = self.button.frame.size;
+     buttonSize.height += 20;
+     buttonSize.width += 20;
+     CGFloat blockX = CGRectGetWidth(self.textView.bounds) - buttonSize.width;
+     CGRect areaToBlockText = CGRectMake(blockX, 0, buttonSize.width, buttonSize.height);
+     UIBezierPath *buttonPath = [UIBezierPath bezierPathWithRect:areaToBlockText];
+
+     self.textView.textContainer.exclusionPaths = @[buttonPath];
+ }
```

The background colors and each view's frame is updated based on the current value of **`isWritingComment`**.

After the conditionals, we make a **`CGRect`** that's a bit larger than the comment button. We convert this into a **`UIBezierPath`**, and add this to the text view's text container's exclusion paths. This means that the text view won't draw text that intersects with this path, causing it to wrap




We're almost done with the view; we just need to implement the rest of the code from the interface.

If we're told to stop editing, dismiss the keyboard:

BLCComposeCommentView.m

```
+ - (void) stopComposingComment {
+     [self.textView resignFirstResponder];
+ }
```

If **`isWritingComment`** changes, update the view appropriately:

BLCComposeCommentView.m

```
+
+ - (void) setIsWritingComment:(BOOL)isWritingComment {
+     [self setIsWritingComment:isWritingComment animated:NO];
+ }
+
+ - (void) setIsWritingComment:(BOOL)isWritingComment animated:(BOOL)animated {
+     _isWritingComment = isWritingComment;
+
+     if (animated) {
+         [UIView animateWithDuration:0.2 animations:^{
+             [self layoutSubviews];
+         }];
+     } else {
+         [self layoutSubviews];
+     }
+ }
```

Here we're providing both an animated and immediate way to update the view. Setting **`isWritingComment`** directly calls **`setIsWritingComment:animated:`**, passing **`NO`** for the **`animated`** variable.

Within **`setIsWritingComment:animated:`**, we store the new setting, and call **`layoutSubviews`** to update the view positioning; either with or without an animation block, as appropriate.

When **`text`** is set, we should update the text view. We should also determine which mode is correct depending on if there's any text:

BLCComposeCommentView.m

```
+ - (void) setText:(NSString *)text {
+     _text = text;
+     self.textView.text = text;
+     self.textView.userInteractionEnabled = YES;
+     self.isWritingComment = text.length > 0;
+ }
```

We also reset **`userInteractionEnabled`** to **`YES`**. (We'll set it to **`NO`** when uploading a comment to the API, so this will ensure the text field is enabled if the user scrolls to another cell.)

When the button is pressed, we want to do one of two things:

1. If the user hasn't started writing, bring up the keyboard.
2. If the user is done writing, send the comment to the API.

BLCComposeCommentView.m

```
+ #pragma mark - Button Target
+
+ - (void) commentButtonPressed:(UIButton *) sender {
+     if (self.isWritingComment) {
+         [self.textView resignFirstResponder];
+         self.textView.userInteractionEnabled = NO;
+         [self.delegate commentViewDidPressCommentButton:self];
+     } else {
+         [self setIsWritingComment:YES animated:YES];
+         [self.textView becomeFirstResponder];
+     }
+ }
```

We'll use the **`UITextViewDelegate`** protocol to inform the delegate of user actions, and to update **`isWritingComment`** appropriately:

BLCComposeCommentView.m

```
+
+ - (BOOL)textViewShouldBeginEditing:(UITextView *)textView {
+     [self setIsWritingComment:YES animated:YES];
+     [self.delegate commentViewWillStartEditing:self];
+
+     return YES;
+ }
+
+ - (BOOL)textView:(UITextView *)textView shouldChangeTextInRange:(NSRange)range replacementText:(NSString *)text {
+     NSString *newText = [textView.text stringByReplacingCharactersInRange:range withString:text];
+     [self.delegate commentView:self textDidChange:newText];
+     return YES;
+ }
+
+ - (BOOL)textViewShouldEndEditing:(UITextView *)textView {
+     BOOL hasComment = (textView.text.length > 0);
+     [self setIsWritingComment:hasComment animated:YES];
+
+     return YES;
+ }
```

All of these delegate methods allow us to restrict the user in some way, but we're not interested in that. In all cases, we use the methods to update state and communicate user intent, but we always return **YES** to allow the user to do what they want.

# Updating the Media Cell

We need to update **BLCMediaTableViewCell** to include this view.

Let's add a property to **BLCMedia** to store the comment as it's being written:

BLCMedia.h

```
@property (nonatomic, assign) BLCLikeState likeState;

+ @property (nonatomic, strong) NSString *temporaryComment;
+
- (instancetype) initWithDictionary:(NSDictionary *)mediaDictionary;
```

In the table cell's **.h** file, let's add the relevant delegate methods:

BLCMediaTableViewCell.h

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

- @class BLCMedia, BLCMediaTableViewCell;
+ @class BLCMedia, BLCMediaTableViewCell, BLCComposeCommentView;

@protocol BLCMediaTableViewCellDelegate <NSObject>

- (void) cell:(BLCMediaTableViewCell *)cell didTapImageView:(UIImageView *)imageView;
- (void) cell:(BLCMediaTableViewCell *)cell didLongPressImageView:(UIImageView *)imageView;
- (void) cellDidPressLikeButton:(BLCMediaTableViewCell *)cell;
+ - (void) cellWillStartComposingComment:(BLCMediaTableViewCell *)cell;
+ - (void) cell:(BLCMediaTableViewCell *)cell didComposeComment:(NSString *)comment;

@end
```

Additionally, let's add a public, **readonly** property for the comment view and a similar **stopComposingComment** method:

BLCMediaTableViewCell.h

```
@property (nonatomic, weak) id <BLCMediaTableViewCellDelegate> delegate;
+ @property (nonatomic, strong, readonly) BLCComposeCommentView *commentView;

+ (CGFloat) heightForMediaItem:(BLCMedia *)mediaItem width:(CGFloat)width;

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

In the **.m** file, we'll need to import it, and declare that we conform to its delegate:

BLCMediaTableViewCell.m

```
#import "BLCComment.h"
#import "BLCUser.h"
#import "BLCLikeButton.h"
+ #import "BLCComposeCommentView.h"

- @interface BLCMediaTableViewCell () <UIGestureRecognizerDelegate>
+ @interface BLCMediaTableViewCell () <UIGestureRecognizerDelegate, BLCComposeCommentViewDelegate>

@property (nonatomic, strong) UIImageView *mediaImageView;
@property (nonatomic, strong) UILabel *usernameAndCaptionLabel;
```

We also need to redeclare the property as writable in the **.m** file, since the one in the **.h** file was **readwrite**:

BLCMediaTableViewCell.m

```
@property (nonatomic, strong) UILongPressGestureRecognizer *longPressGestureRecognizer;

@property (nonatomic, strong) BLCLikeButton *likeButton;
+ @property (nonatomic, strong) BLCComposeCommentView *commentView;

@end
```

In the initializer, we'll create the comment view and add it to view with constraints:

BLCMediaTableViewCell.m

```
        [self.likeButton addTarget:self action:@selector(likePressed:) forControlEvents:UIControlEventTouchUpInside];
        self.likeButton.backgroundColor = usernameLabelGray;

+         self.commentView = [[BLCComposeCommentView alloc] init];
+         self.commentView.delegate = self;
+
-         for (UIView *view in @[self.mediaImageView, self.usernameAndCaptionLabel, self.commentLabel, self.likeButton]) {
+         for (UIView *view in @[self.mediaImageView, self.usernameAndCaptionLabel, self.commentLabel, self.likeButton, self.commentV
            [self.contentView addSubview:view];
            view.translatesAutoresizingMaskIntoConstraints = NO;
        }

-         NSDictionary *viewDictionary = NSDictionaryOfVariableBindings(_mediaImageView, _usernameAndCaptionLabel, _commentLabel, _li
+         NSDictionary *viewDictionary = NSDictionaryOfVariableBindings(_mediaImageView, _usernameAndCaptionLabel, _commentLabel, _li

        [self.contentView addConstraints:[NSLayoutConstraint constraintsWithVisualFormat:@"H:|[_mediaImageView]|" options:kNilOptions
        [self.contentView addConstraints:[NSLayoutConstraint constraintsWithVisualFormat:@"H:|[_usernameAndCaptionLabel][_likeButton(
        [self.contentView addConstraints:[NSLayoutConstraint constraintsWithVisualFormat:@"H:|[_commentLabel]|" options:kNilOptions m
+         [self.contentView addConstraints:[NSLayoutConstraint constraintsWithVisualFormat:@"H:|[_commentView]|" options:kNilOptions

-         [self.contentView addConstraints:[NSLayoutConstraint constraintsWithVisualFormat:@"V:|[_mediaImageView][_usernameAndCaption
+         [self.contentView addConstraints:[NSLayoutConstraint constraintsWithVisualFormat:@"V:|[_mediaImageView][_usernameAndCaption
                                                                                  options:kNilOptions
                                                                                  metrics:nil
                                                                                    views:viewDictionary]];
```

When calculating height, we should now use the bottom of the comment view instead of the comment label. (Remember that the height is

BLCMediaTableViewCell.m

```
    [layoutCell layoutIfNeeded];

    // Get the actual height required for the cell
-     return CGRectGetMaxY(layoutCell.commentLabel.frame);
+     return CGRectGetMaxY(layoutCell.commentView.frame);
 }
```

When the table cell is created or reused, we need to update the text on the comment view in **`setMediaItem:`**:

BLCMediaTableViewCell.m

```
    self.usernameAndCaptionLabel.attributedText = [self usernameAndCaptionString];
    self.commentLabel.attributedText = [self commentString];
    self.likeButton.likeButtonState = mediaItem.likeState;
+     self.commentView.text = mediaItem.temporaryComment;
 }
```

We also need to implement the delegate methods and **`stopComposingComment`**:

BLCMediaTableViewCell.m

```
+ #pragma mark - BLCComposeCommentViewDelegate
+
+ - (void) commentViewDidPressCommentButton:(BLCComposeCommentView *)sender {
+     [self.delegate cell:self didComposeComment:self.mediaItem.temporaryComment];
+ }
+
+ - (void) commentView:(BLCComposeCommentView *)sender textDidChange:(NSString *)text {
+     self.mediaItem.temporaryComment = text;
+ }
+
+ - (void) commentViewWillStartEditing:(BLCComposeCommentView *)sender {
+     [self.delegate cellWillStartComposingComment:self];
+ }
+
+ - (void) stopComposingComment {
+     [self.commentView stopComposingComment];
+ }
```

In both **`commentViewDidPressCommentButton:`** and **`commentViewWillStartEditing:`**, the cell simply tells its delegate (the images table controller) that a comment was composed or that the user began editing.

In **`commentView:textDidChange:`**, we just hang on to the text written so far in **`temporaryComment`**. This allows a user to scroll up and down the table without losing any partially written comments.

If another object tells the cell to **`stopComposingComment`**, the cell passes that message along to the comment view.

# Updating the Data Source

We'll need to add a method to the data source to handle comments.

In the **`.h`** file:

BLCDataSource.h

```
- (void) toggleLikeOnMediaItem:(BLCMedia *)mediaItem;
+ - (void) commentOnMediaItem:(BLCMedia *)mediaItem withCommentText:(NSString *)commentText;

@end
```

The implementation will post the method to **Instagram's comments endpoint**. If it succeeds, it'll refetch the media item with the new comment. If not, it simply reloads the row:

BLCDataSource.m

```
+ #pragma mark - Comments
+
+ - (void) commentOnMediaItem:(BLCMedia *)mediaItem withCommentText:(NSString *)commentText {
+     if (!commentText || commentText.length == 0) {
+         return;
+     }
+
+     NSString *urlString = [NSString stringWithFormat:@"media/%@/comments", mediaItem.idNumber];
+     NSDictionary *parameters = @{@"access_token": self.accessToken, @"text": commentText};
+
+     [self.instagramOperationManager POST:urlString parameters:parameters success:^(AFHTTPRequestOperation *operation, id responseOb
+         mediaItem.temporaryComment = nil;
+
+         NSString *refreshMediaUrlString = [NSString stringWithFormat:@"media/%@", mediaItem.idNumber];
+         NSDictionary *parameters = @{@"access_token": self.accessToken};
+         [self.instagramOperationManager GET:refreshMediaUrlString parameters:parameters success:^(AFHTTPRequestOperation *operation
+             BLCMedia *newMediaItem = [[BLCMedia alloc] initWithDictionary:responseObject];
+             NSMutableArray *mutableArrayWithKVO = [self mutableArrayValueForKey:@"mediaItems"];
+             NSUInteger index = [mutableArrayWithKVO indexOfObject:mediaItem];
+             [mutableArrayWithKVO replaceObjectAtIndex:index withObject:newMediaItem];
+         } failure:^(AFHTTPRequestOperation *operation, NSError *error) {
+             [self reloadMediaItem:mediaItem];
+         }];
+     } failure:^(AFHTTPRequestOperation *operation, NSError *error) {
+         NSLog(@"Error: %@", error);
+         NSLog(@"Response: %@", operation.responseString);
+         [self reloadMediaItem:mediaItem];
+     }];
+ }

@end
```

(As noted at the beginning of the checkpoint, the failure block will get called unless you've obtained special commenting permission from Instagram.)

# Updating the Images Table Controller

We'll update the images table controller.

Add properties for the comment view and keyboard height adjustments:

BLCImagesTableViewController.m

```
@interface BLCImagesTableViewController () <BLCMediaTableViewCellDelegate, UIViewControllerTransitioningDelegate>

@property (nonatomic, weak) UIImageView *lastTappedImageView;
+ @property (nonatomic, weak) UIView *lastSelectedCommentView;
+ @property (nonatomic, assign) CGFloat lastKeyboardAdjustment;

@end
```

In **viewDidLoad**, we'll request notifications of the keyboard appearing and disappearing. We'll also enable

```
    [self.refreshControl addTarget:self action:@selector(refreshControlDidFire:) forControlEvents:UIControlEventValueChanged];

    [self.tableView registerClass:[BLCMediaTableViewCell class] forCellReuseIdentifier:@"mediaCell"];
+
+     self.tableView.keyboardDismissMode = UIScrollViewKeyboardDismissModeInteractive;
+
+     [[NSNotificationCenter defaultCenter] addObserver:self
+                                              selector:@selector(keyboardWillShow:)
+                                                  name:UIKeyboardWillShowNotification
+                                                object:nil];
+
+     [[NSNotificationCenter defaultCenter] addObserver:self
+                                              selector:@selector(keyboardWillHide:)
+                                                  name:UIKeyboardWillHideNotification
+                                                object:nil];
+ }
```

These notifications will call **`keyboardWillShow:`** and **`keyboardWillHide:`** (which we'll implement in a bit) before the keyboard shows or hides.

Just like key-value observing, we should unregister for these notifications in **`dealloc`**:

BLCImagesTableViewController.m

```
 - (void) dealloc {
    [[BLCDataSource sharedInstance] removeObserver:self forKeyPath:@"mediaItems"];
+     [[NSNotificationCenter defaultCenter] removeObserver:self];
 }
```

**`UITableViewController`** has some built-in logic for handling keyboard appearance, but it doesn't work on large cells like ours. This logic is implemented in its **`viewWillAppear:`** and **`viewWillDisappear:`** methods. We want to implement our own logic, so we'll disable this functionality by overriding these two methods without calling **`super`**.

We do, however, want to keep a different piece of functionality - making sure the cells aren't selected when the view appears. Since we're skipping the code that does that, we'll now need to do that ourself:

BLCImagesTableViewController.m

```
+ - (void)viewWillAppear:(BOOL)animated {
+     NSIndexPath *indexPath = self.tableView.indexPathForSelectedRow;
+     if (indexPath) {
+         [self.tableView deselectRowAtIndexPath:indexPath animated:animated];
+     }
+ }
+
+ - (void) viewWillDisappear:(BOOL)animated {
+
+ }
```

To minimize jerky scrolling, let's account for the comment view in the height estimation:

BLCImagesTableViewController.m

```
 - (CGFloat) tableView:(UITableView *)tableView estimatedHeightForRowAtIndexPath:(NSIndexPath *)indexPath {
    BLCMedia *item = [BLCDataSource sharedInstance].mediaItems[indexPath.row];
    if (item.image) {
-         return 350;
+         return 450;
    } else {
-         return 150;
+         return 250;
    }
 }
```

If a row is tapped, let's assume the user doesn't want the keyboard:

```
+ - (void) tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
+     BLCMediaTableViewCell *cell = (BLCMediaTableViewCell *)[tableView cellForRowAtIndexPath:indexPath];
+     [cell stopComposingComment];
+ }
 - (void) tableView:(UITableView *)tableView willDisplayCell:(UITableViewCell *)cell forRowAtIndexPath:(NSIndexPath *)indexPath {
    BLCMedia *mediaItem = [BLCDataSource sharedInstance].mediaItems[indexPath.row];
```

We need to respond to the new delegate methods we created:

BLCImagesTableViewController.m

```
    [[BLCDataSource sharedInstance] toggleLikeOnMediaItem:cell.mediaItem];
}

+ - (void) cellWillStartComposingComment:(BLCMediaTableViewCell *)cell {
+     self.lastSelectedCommentView = (UIView *)cell.commentView;
+ }
+
+ - (void) cell:(BLCMediaTableViewCell *)cell didComposeComment:(NSString *)comment {
+     [[BLCDataSource sharedInstance] commentOnMediaItem:cell.mediaItem withCommentText:comment];
+ }
+
#pragma mark - UIViewControllerTransitioningDelegate

- (id<UIViewControllerAnimatedTransitioning>)animationControllerForPresentedController:(UIViewController *)presented
```

If the user starts composing a comment, we'll save a reference to the comment view. If the user presses the comment button, we'll tell the AP to send a comment.

# Handling Keyboard Events

We still need to implement the two keyboard methods, **`keyboardWillShow:`** and **`keyboardWillHide:`**.

When the keyboard shows, we need to do a few things:

- We need to determine if the keyboard will obscure the comment view's text box. (This will happen if the user taps a comment on the lower half of the screen).
- If the keyboard will obscure the comment view, adjust the table view's content insets and offset to move the comment view directly above the keyboard.
- If we make any changes, save them for later so we can undo them when the keyboard hides.

Here's how that looks in code:

BLCImagesTableViewController.m

```
}

+ #pragma mark - Keyboard Handling
+
+ - (void)keyboardWillShow:(NSNotification *)notification
+ {
+     // Get the frame of the keyboard within self.view's coordinate system
+     NSValue *frameValue = notification.userInfo[UIKeyboardFrameEndUserInfoKey];
+     CGRect keyboardFrameInScreenCoordinates = frameValue.CGRectValue;
+     CGRect keyboardFrameInViewCoordinates = [self.navigationController.view convertRect:keyboardFrameInScreenCoordinates fromView:n
+
+     // Get the frame of the comment view in the same coordinate system
+     CGRect commentViewFrameInViewCoordinates = [self.navigationController.view convertRect:self.lastSelectedCommentView.bounds from
+
+     CGPoint contentOffset = self.tableView.contentOffset;
+     UIEdgeInsets contentInsets = self.tableView.contentInset;
+     UIEdgeInsets scrollIndicatorInsets = self.tableView.scrollIndicatorInsets;
+     CGFloat heightToScroll = 0;
+
+     CGFloat keyboardY = CGRectGetMinY(keyboardFrameInViewCoordinates);
+     CGFloat commentViewY = CGRectGetMinY(commentViewFrameInViewCoordinates);
+     CGFloat difference = commentViewY - keyboardY;
+
+     if (difference > 0) {
+         heightToScroll += difference;
+     }
+
+     if (CGRectIntersectsRect(keyboardFrameInViewCoordinates, commentViewFrameInViewCoordinates)) {
+         // The two frames intersect (the keyboard would block the view)
+         CGRect intersectionRect = CGRectIntersection(keyboardFrameInViewCoordinates, commentViewFrameInViewCoordinates);
+         heightToScroll += CGRectGetHeight(intersectionRect);
+     }
+
+     if (heightToScroll > 0) {
+         contentInsets.bottom += heightToScroll;
+         scrollIndicatorInsets.bottom += heightToScroll;
+         contentOffset.y += heightToScroll;
+
+         NSNumber *durationNumber = notification.userInfo[UIKeyboardAnimationDurationUserInfoKey];
+         NSNumber *curveNumber = notification.userInfo[UIKeyboardAnimationCurveUserInfoKey];
+
+         NSTimeInterval duration = durationNumber.doubleValue;
+         UIViewAnimationCurve curve = curveNumber.unsignedIntegerValue;
+         UIViewAnimationOptions options = curve << 16;
+
+         [UIView animateWithDuration:duration delay:0 options:options animations:^{
+             self.tableView.contentInset = contentInsets;
+             self.tableView.scrollIndicatorInsets = scrollIndicatorInsets;
+             self.tableView.contentOffset = contentOffset;
+         } completion:nil];
+     }
+
+     self.lastKeyboardAdjustment = heightToScroll;
+ }
```

Let's step through the code. We get the keyboard's frame and the comment view's frame, and make sure they're in the same coordinate system. If the keyboard would be higher on the screen than the comment view, we account for the positioning difference. If the two frames intersect, we adjust the height by the amount of their intersection. Finally, we animate the changes, and store the adjustment for future use.

When the keyboard hides, we simply reverse these changes:

BLCImagesTableViewController.m

```
+ {
+     UIEdgeInsets contentInsets = self.tableView.contentInset;
+     contentInsets.bottom -= self.lastKeyboardAdjustment;
+
+     UIEdgeInsets scrollIndicatorInsets = self.tableView.scrollIndicatorInsets;
+     scrollIndicatorInsets.bottom -= self.lastKeyboardAdjustment;
+
+     NSNumber *durationNumber = notification.userInfo[UIKeyboardAnimationDurationUserInfoKey];
+     NSNumber *curveNumber = notification.userInfo[UIKeyboardAnimationCurveUserInfoKey];
+
+     NSTimeInterval duration = durationNumber.doubleValue;
+     UIViewAnimationCurve curve = curveNumber.unsignedIntegerValue;
+     UIViewAnimationOptions options = curve << 16;
+
+     [UIView animateWithDuration:duration delay:0 options:options animations:^{
+         self.tableView.contentInset = contentInsets;
+         self.tableView.scrollIndicatorInsets = scrollIndicatorInsets;
+     } completion:nil];
+ }
```

Run the app. You should now be able to write comments in the text box on each media item, as shown in the screen shot.



Our current solution isn't ***indempotent***. It assumes that **`keyboardWillShow:`** and **`keyboardWillHide:`** will be called equally and in balance. However, this is false: if we rotate the device, **`keyboardWillShow:`** is called, but **`keyboardWillHide:`**isn't.

Continuously rotating the device can result in the content inset and offset becoming increasingly inaccurate.

Update **`keyboardWillShow:`** and **`keyboardWillHide:`** to be indempotent, and to properly handle rotation. You'll need to update the same variables based off their initial values, not their current values, since the current values may already be included when **`keyboardWillShow:`** starts.

This is a more challenging assignment; you may want to pair program with your mentor to work on it.

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 Comment View'
$ git checkout master
$ git merge comment-view
$ 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

Roadmap · Previous Checkpoint
Jump to Submission · Next Checkpoir

# Taking Pictures


In this checkpoint, you'll build a view controller that lets the user take a square picture.

In this process you'll learn a lot of new stuff!

- You'll learn the basics of **the `AVFoundation` framework**, a powerful framework for playing, saving, editing and encoding photos, videos and sound.
- You'll learn how to use **Objective-C Categories** to extend the functionality of a class.
- You'll learn some **Core Graphics** functionality for rotating, resizing, and cropping images.

## Building the Views

Before we can do cool stuff with images, let's put together some views for users to interact with

Terminal

```
$ git checkout -b taking-pictures
```

## Toolbar View

First we're going to make a toolbar view. The code structure will be very similar to the toolbar you built in the web browser checkpoints.

**Download these resources** and drag them into your **`Images.xcassets`** file (just like you did with the heart icon earlier.)

This view will look very familiar. Here's the **`.h`** file:

BLCCameraToolbar.h

```
+ @class BLCCameraToolbar;
+
+ @protocol BLCCameraToolbarDelegate <NSObject>
+
+ - (void) leftButtonPressedOnToolbar:(BLCCameraToolbar *)toolbar;
+ - (void) rightButtonPressedOnToolbar:(BLCCameraToolbar *)toolbar;
+ - (void) cameraButtonPressedOnToolbar:(BLCCameraToolbar *)toolbar;
+
+ @end
+
 @interface BLCCameraToolbar : UIView

+ - (instancetype) initWithImageNames:(NSArray *)imageNames;
+
+ @property (nonatomic, weak) NSObject <BLCCameraToolbarDelegate> *delegate;
+
 @end
```

As you may have guessed, our toolbar will have three buttons: customizable left and right buttons, and a center button that looks like a camera.

The image names for the icons on the side buttons will be passed in our initializer, **`initWithImageNames:`**.

The view will know nearly nothing about what the function of these buttons is. Instead, we'll use the trusted delegate pattern to inform the view when the buttons are pressed.

Let's take a look at the **`.m`** file.

First, we'll start by declaring properties for the three buttons, plus two views we'll use to customize the appearance of the view:

BLCCameraToolbar.m

```
+ @interface BLCCameraToolbar ()
+
+ @property (nonatomic, strong) UIButton *leftButton;
+ @property (nonatomic, strong) UIButton *cameraButton;
+ @property (nonatomic, strong) UIButton *rightButton;
+
+ @property (nonatomic, strong) UIView *whiteView;
+ @property (nonatomic, strong) UIView *purpleView;
+
+ @end
```

In the initializer, we'll create all of these objects and add them to the view hierarchy. We add targets for the buttons. We have a default image name (**`camera`**) for the camera button; the other two images will be passed in us.

BLCCameraToolbar.m

```
+ - (instancetype) initWithImageNames:(NSArray *)imageNames {
+     self = [super init];
+
+     if (self) {
+         self.leftButton = [UIButton buttonWithType:UIButtonTypeCustom];
+         self.cameraButton = [UIButton buttonWithType:UIButtonTypeCustom];
+         self.rightButton = [UIButton buttonWithType:UIButtonTypeCustom];
+
+         [self.leftButton addTarget:self action:@selector(leftButtonPressed:) forControlEvents:UIControlEventTouchUpInside];
+         [self.cameraButton addTarget:self action:@selector(cameraButtonPressed:) forControlEvents:UIControlEventTouchUpInside];
+         [self.rightButton addTarget:self action:@selector(rightButtonPressed:) forControlEvents:UIControlEventTouchUpInside];
+
+         [self.leftButton setImage:[UIImage imageNamed:imageNames.firstObject] forState:UIControlStateNormal];
+
+         [self.rightButton setImage:[UIImage imageNamed:imageNames.lastObject] forState:UIControlStateNormal];
+
+         [self.cameraButton setImage:[UIImage imageNamed:@"camera"] forState:UIControlStateNormal];
+         [self.cameraButton setContentEdgeInsets:UIEdgeInsetsMake(10, 10, 15, 10)];
+
+         self.whiteView = [UIView new];
+         self.whiteView.backgroundColor = [UIColor whiteColor];
+
+         self.purpleView = [UIView new];
+         self.purpleView.backgroundColor = [UIColor colorWithRed:0.345 green:0.318 blue:0.424 alpha:1]; /*#58516c*/
+
+         for (UIView *view in @[self.whiteView, self.purpleView, self.leftButton, self.cameraButton, self.rightButton]) {
+             [self addSubview:view];
+         }
+     }
+
+     return self;
+ }
```

In **`layoutSubviews`**, we'll determine the positioning of these items. They'll be spread out evenly across the horizontal axis. We'll also use the **`CAShapeLayer`** and **`UIBezierPath`** classes we learned about earlier to round the top left and top right corners of the purple view behind the camera button.

BLCCameraToolbar.m

```
+ - (void) layoutSubviews {
+     [super layoutSubviews];
+
+     CGRect whiteFrame = self.bounds;
+     whiteFrame.origin.y += 10;
+     self.whiteView.frame = whiteFrame;
+
+     CGFloat buttonWidth = CGRectGetWidth(self.bounds) / 3;
+
+     NSArray *buttons = @[self.leftButton, self.cameraButton, self.rightButton];
+     for (int i = 0; i < 3; i++) {
+         UIButton *button = buttons[i];
+         button.frame = CGRectMake(i * buttonWidth, 10, buttonWidth, CGRectGetHeight(whiteFrame));
+     }
+
+     self.purpleView.frame = CGRectMake(buttonWidth, 0, buttonWidth, CGRectGetHeight(self.bounds));
+
+     UIBezierPath *maskPath = [UIBezierPath bezierPathWithRoundedRect:self.purpleView.bounds
+                                                    byRoundingCorners:UIRectCornerTopLeft | UIRectCornerTopRight
+                                                          cornerRadii:CGSizeMake(10.0, 10.0)];
+
+     CAShapeLayer *maskLayer = [CAShapeLayer layer];
+     maskLayer.frame = self.purpleView.bounds;
+     maskLayer.path = maskPath.CGPath;
+
+     self.purpleView.layer.mask = maskLayer;
+ }
```

BLCCameraToolbar.m

```
+ # pragma mark - Button Handlers
+
+ - (void) leftButtonPressed:(UIButton *)sender {
+     [self.delegate leftButtonPressedOnToolbar:self];
+ }
+
+ - (void) rightButtonPressed:(UIButton *)sender {
+     [self.delegate rightButtonPressedOnToolbar:self];
+ }
+
+ - (void) cameraButtonPressed:(UIButton *)sender {
+     [self.delegate cameraButtonPressedOnToolbar:self];
+ }
@end
```

Creating views and using the delegate pattern should be starting to look familiar to you. By following the patterns we've learned and established, we're able to quickly build useful and functional views.

## Camera View Controller

We'll write a camera view controller which uses this toolbar. Users will use this view controller to take pictures. In later checkpoints, we'll appl filters and allow the user to send the image to the Instagram app for upload. (Instagram doesn't allow posting pictures via their third-party API.)

The interface will look simple:

BLCCameraViewController.h

```
+ @class BLCCameraViewController;
+
+ @protocol BLCCameraViewControllerDelegate <NSObject>
+
+ - (void) cameraViewController:(BLCCameraViewController *)cameraViewController didCompleteWithImage:(UIImage *)image;
+
+ @end
+
@interface BLCCameraViewController : UIViewController

+ @property (nonatomic, weak) NSObject <BLCCameraViewControllerDelegate> *delegate;
+
+ @end
```

All we have is a delegate property and accompanying protocol to inform the presenting view controller when the camera view controller is done.

Let's look at the **.m** file. First, we're going to import the **AVFoundation** framework as well as the toolbar we made:

BLCCameraViewController.m

```
#import "BLCCameraViewController.h"

+ #import <AVFoundation/AVFoundation.h>
+
+ #import "BLCCameraToolbar.h"
```

Now add the properties and declare which delegates **BLCCameraViewController** conforms to:

BLCCameraViewController.m

```
+ @interface BLCCameraViewController () <BLCCameraToolbarDelegate, UIAlertViewDelegate>
+
+ @property (nonatomic, strong) UIView *imagePreview;
+
+ @property (nonatomic, strong) AVCaptureSession *session;
+ @property (nonatomic, strong) AVCaptureVideoPreviewLayer *captureVideoPreviewLayer;
+ @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) BLCCameraToolbar *cameraToolbar;

@end
```

Let's go through these objects and briefly describe what they're for. The objects beginning with **`AVCapture...`** are new to you, and you'll learn how to use them throughout this checkpoint.

**`imagePreview`** will be a view that shows the user what the camera is currently pointing at.

**`session`** is an **`AVCaptureSession` object**. Capture sessions coordinate data from inputs (cameras and microphones) to the outputs (movie files and still images).

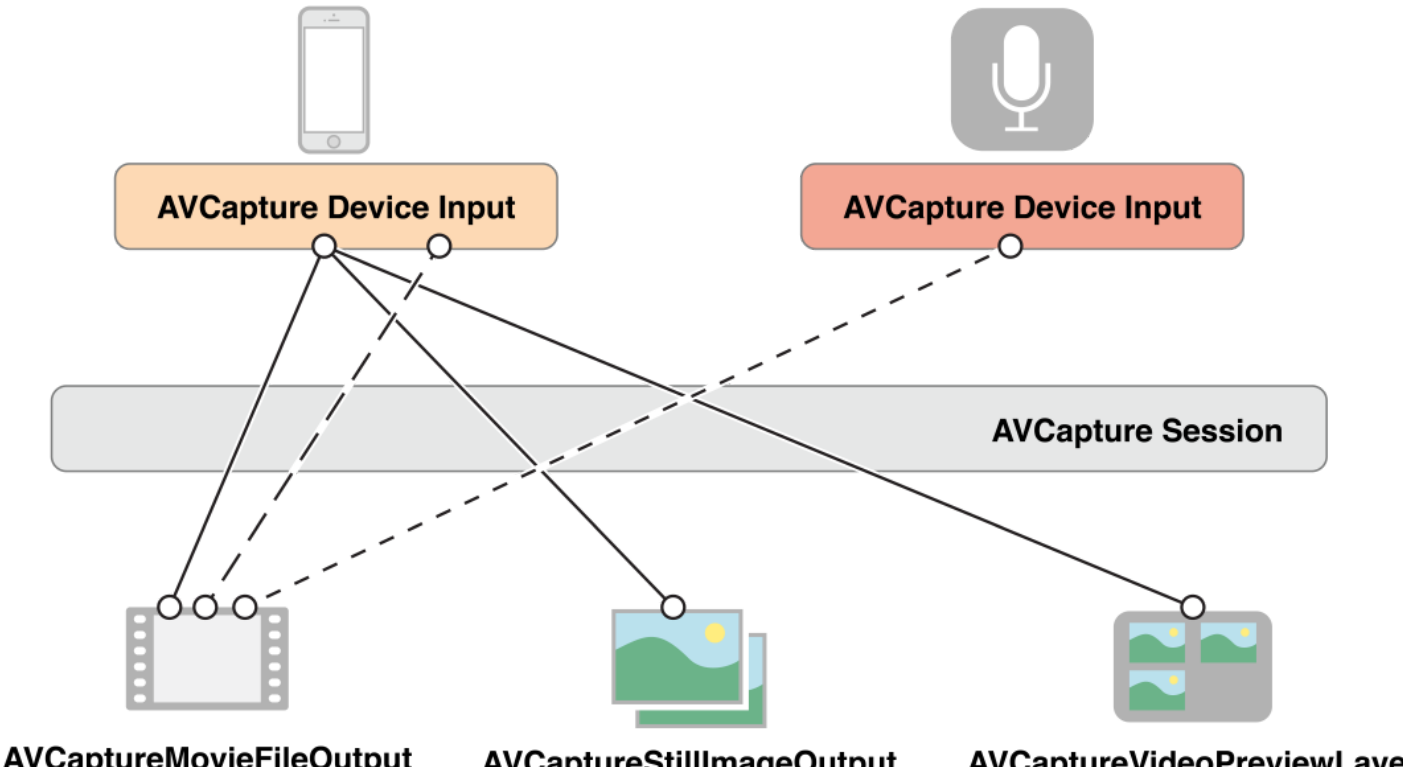
**`captureVideoPreviewLayer`** is a special type of **`CALayer`**. You may remember **`CALayer`** from the checkpoint in which we created a like button:

> *Visual content is represented by **layers**. These are represented by the **`CALayer`** class (or subclasses like **`CAShapeLayer`**). You already use these: **`UIView`** translates everything you do to **`CALayer`**s under the hood.*

**`AVCaptureVideoPreviewLayer`** displays video from a camera.

**`stillImageOutput`** captures high-quality still images from the capture session's input (camera).

Here's a diagram of how these **`AVCapture...`** objects work together:



capture area.

**topView** and **bottomView** are **UIToolbar** objects. Toolbars are typically used for adding small buttons to a view. In this case, however, we're just using them for their unique translucent effect. (We won't be adding any buttons to them.)

**cameraToolbar** will store the camera toolbar we created earlier in this checkpoint.

## Initial View Setup

Our initial setup of this view will be a bit longer than normal, so we'll split **viewDidLoad** into 4 sections:

1. Creating all of the views
2. Adding the views to the view hierarchy
3. Setting up Image Capture
4. Creating a **Cancel** button in the toolbar

## Creating the Subviews

Let's start with creating the views:

BLCCameraViewController.m

```
+ #pragma mark - Build View Hierarchy
+
+ - (void)viewDidLoad {
+     [super viewDidLoad];
+     // Do any additional setup after loading the view.
+
+     [self createViews];
+ }
+
+ - (void) createViews {
+     self.imagePreview = [UIView new];
+     self.topView = [UIToolbar new];
+     self.bottomView = [UIToolbar new];
+     self.cameraToolbar = [[BLCCameraToolbar alloc] initWithImageNames:@[@"rotate", @"road"]];
+     self.cameraToolbar.delegate = self;
+     UIColor *whiteBG = [UIColor colorWithWhite:1.0 alpha:.15];
+     self.topView.barTintColor = whiteBG;
+     self.bottomView.barTintColor = whiteBG;
+     self.topView.alpha = 0.5;
+     self.bottomView.alpha = 0.5;
+ }
```

**barTintColor** is like **backgroundColor** but it'll be translucent when it's rendered.

## Adding the Subviews to the View Hierarchy

Let's add the views to the view hierarchy:

BLCCameraViewController.m

```
    [super viewDidLoad];
    // Do any additional setup after loading the view.

    [self createViews];
+     [self addViewsToViewHierarchy];
}

+ - (void) addViewsToViewHierarchy {
+     NSMutableArray *views = [@[self.imagePreview, self.topView, self.bottomView] mutableCopy];
+     [views addObjectsFromArray:self.horizontalLines];
+     [views addObjectsFromArray:self.verticalLines];
+     [views addObject:self.cameraToolbar];
+
+     for (UIView *view in views) {
+         [self.view addSubview:view];
+     }
+ }
```

This method adds all of the views. Order is important here: the views added later will be on top.

Currently **`self.horizontalLines`** and **`self.verticalLines`** are **`nil`**, so let's override their getters with some white views:

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;
+ }
```

Great!

## Setting up the Image Capture

Now, let's begin our foray into AV capture sessions:

BLCCameraViewController.m

```
    [super viewDidLoad];
    // Do any additional setup after loading the view.

    [self createViews];
    [self addViewsToViewHierarchy];
+     [self setupImageCapture];
}

+ - (void) setupImageCapture {
+     self.session = [[AVCaptureSession alloc] init];
+     self.session.sessionPreset = AVCaptureSessionPresetHigh;
+
+     self.captureVideoPreviewLayer = [[AVCaptureVideoPreviewLayer alloc] initWithSession:self.session];
+     self.captureVideoPreviewLayer.videoGravity = AVLayerVideoGravityResizeAspectFill;
+     self.captureVideoPreviewLayer.masksToBounds = YES;
+     [self.imagePreview.layer addSublayer:self.captureVideoPreviewLayer];
+
+     [AVCaptureDevice requestAccessForMediaType:AVMediaTypeVideo completionHandler:^(BOOL granted) {
+         dispatch_async(dispatch_get_main_queue(), ^{
+             if (granted) {
+                 AVCaptureDevice *device = [AVCaptureDevice defaultDeviceWithMediaType:AVMediaTypeVideo];
+
+                 NSError *error = nil;
+                 AVCaptureDeviceInput *input = [AVCaptureDeviceInput deviceInputWithDevice:device error:&error];
+                 if (!input) {
+                     UIAlertView *alert = [[UIAlertView alloc] initWithTitle:error.localizedDescription message:error.localizedRecov
+                     [alert show];
+                 } else {
+                     [self.session addInput:input];
+
+                     self.stillImageOutput = [[AVCaptureStillImageOutput alloc] init];
+                     self.stillImageOutput.outputSettings = @{AVVideoCodecKey: AVVideoCodecJPEG};
+
+                     [self.session addOutput:self.stillImageOutput];
+
+                     [self.session startRunning];
+                 }
+             } else {
+                 UIAlertView *alert = [[UIAlertView alloc] initWithTitle:NSLocalizedString(@"Camera Permission Denied", @"camera per
+                                                                 message:NSLocalizedString(@"This app doesn't have permission to use
+                                                                delegate:self
+                                                       cancelButtonTitle:NSLocalizedString(@"OK", @"OK button")
+                                                       otherButtonTitles:nil];
+                 [alert show];
+             }
+         });
+     }];
+ }
```

Most of this is new, so let's step through it. We begin by creating our capture session, which will mediate between the camera and our output layer.

From there, we create **`self.captureVideoPreviewLayer`** to display the camera content. We set **`videoGravity`** to **`AVLayerVideoGravityResizeAspectFill`**. This is equivalent to setting a **`UIImageView`**'s **`contentMode`** property to **`UIViewContentModeScaleAspectFill`**. We call **`addSublayer:`** which is analogous to calling **`addSubview:`** on a **`UIView`**.

We must now receive permission from the user to access the camera by calling **`[AVCaptureDevice +requestAccessForMediaType:completionHandler:]`**. Because the user might take a while to grant or deny permission, we handle the response asynchronously in a completion block. (This is just like how we handle downloading data off the Internet or any potentially long-running work.)

Once inside the completion block, the user has either accepted or declined our access; this is indicated by **`granted`**.

If **`granted == YES`**, then we create a *device* (**`AVCaptureDevice`**). This object represents the camera, and it provides its data to the **`AVCaptureSession`** through an **`AVCaptureDeviceInput`** object, which we create next.

We'll add the input to our capture session, create a still image output that saves JPEG files, and start running the session.

by informing the presenting controller that we'll be unable to get an image:

BLCCameraViewController.m

```
+ #pragma mark - UIAlertViewDelegate
+
+ - (void)alertView:(UIAlertView *)alertView didDismissWithButtonIndex:(NSInteger)buttonIndex {
+     [self.delegate cameraViewController:self didCompleteWithImage:nil];
+ }
```

## Creating a Cancel Button

In case the user doesn't want to take a photo, let's add a cancel button:

BLCCameraViewController.m

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

     [self createViews];
     [self addViewsToViewHierarchy];
     [self setupImageCapture];
+      [self createCancelButton];
 }

+ - (void) createCancelButton {
+     UIImage *cancelImage = [UIImage imageNamed:@"x"];
+     UIBarButtonItem *cancelButton = [[UIBarButtonItem alloc] initWithImage:cancelImage style:UIBarButtonItemStyleDone target:self a
+     self.navigationItem.leftBarButtonItem = cancelButton;
+ }
```

If the button is pressed, inform the delegate:

BLCCameraViewController.m

```
+ #pragma mark - Event Handling
+
+ - (void) cancelPressed:(UIBarButtonItem *)sender {
+     [self.delegate cameraViewController:self didCompleteWithImage:nil];
+ }
```

## Laying out the Subviews

The top and bottom views should be present over the areas of the photo that won't be saved.

The horizontal and vertical lines are distributed evenly over the photo area to create a 3x3 grid of squares.

**`self.imagePreview`** and **`self.captureVideoPreviewLayer`** are configured to take up all of the space in the view controller's primary view.

Finally, we layout the camera toolbar at the bottom.

BLCCameraViewController.m

```
+
+ - (void)viewWillLayoutSubviews {
+     [super viewWillLayoutSubviews];
+
+     CGFloat width = CGRectGetWidth(self.view.bounds);
+     self.topView.frame = CGRectMake(0, self.topLayoutGuide.length, width, 44);
+
+     CGFloat yOriginOfBottomView = CGRectGetMaxY(self.topView.frame) + width;
+     CGFloat heightOfBottomView = CGRectGetHeight(self.view.frame) - yOriginOfBottomView;
+     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.imagePreview.frame = self.view.bounds;
+     self.captureVideoPreviewLayer.frame = self.imagePreview.bounds;
+
+     CGFloat cameraToolbarHeight = 100;
+     self.cameraToolbar.frame = CGRectMake(0, CGRectGetHeight(self.view.bounds) - cameraToolbarHeight, width, cameraToolbarHeight);
+ }
```

All that's left for this view controller is responding to the three button taps! Sounds easy, right? Well, it's a little more complex than it looks. This process will teach you a lot more about how iOS images work under the hood, and will give you a new appreciation for photo sharing apps like Instagram and Facebook.

## Responding to the Left Camera Toolbar Button

The left button is intended to flip between the front and rear cameras.

BLCCameraViewController.m