You should now have an iOS app that scrolls a bunch of images up and down!

Your assignment:

- Make your image rows *deletable* by implementing `tableView:canEditRowAtIndexPath:` and `tableView:commitEditingStyle:forRowAtIndexPath:`
  - The comments within the pre-written implementation should give you some hints as to how you should accomplish this
  - Remember to update your image array as well to actually delete it from the collection
  - Rows are editable by swiping left on them

Since this is a new project, remember to first create a new repo in GitHub. You can name it "blocstagram" - as we do below - or any other name that's to your liking.

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

Terminal

```
$ git add .
$ git commit -m 'Table View controller implemented'
$ git checkout master
$ git merge building-blocstagram
$ git remote add origin https://github.com/<username>/blocstagram.git
$ git push -u origin master
```

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

assignment completed

Roadmap · Previous Checkpoint Jump to Submission · Next Checkpoir

Refactoring User Names and Captions

Currently, our table view's data source - our model - is the **`NSMutableArray`** called **`images`**. It is created and populated with images by our view controller.

This is a poor design choice. We've combined our **model** (the images) and our **controller** (the view controller) into one class. This will make it difficult if we want to populate the images from some other place. Additionally, as we add in more code for features like downloading images viewing comments from different users, etc., our view controller will become complex. By the end of this checkpoint the app will continue to behave the same way it does now. It will look exactly the same from a user's perspective, but it will be much different under the hood. This process is called **refactoring**: the reorganization of code without changing any outwardly visible behavior.

This is something that all good programmers must do to keep apps from becoming too disorganized. Messy code is impossible to work with efficiently.

For the first step in this refactor, we'll be introducing the concept of a **model**. iOS is organized around a programming pattern called **Model View Controller** or **MVC**. We've already been using views throughout the previous checkpoints.

A **model** is meant to be a representation of data. When writing an application using MVC we generally create a model for each kind of data object in our system. For example, if you are making a social photography app, you might have three separate models to represent a **user**, a **photograph** and a **comment**. Models sometimes also fetch and store the data they manage.

To summarize, MVC is a pattern for organizing applications that centers around three kinds of objects:

- **Models** - representations of data (knowledge)
- **Views** - responsible for displaying data
- **Controllers** - responsible for mediating between models and views as well as responding to events.

Please review **iOS Design Patterns: Delegation & Pro** ? and discuss it with your mentor.

For Blocstagram, we'll make three new data model objects:

- An object representing one user, **`BLCUser`**.
- An object representing one image post, **`BLCMedia`**.
- An object representing one comment, **`BLCComment`**.

We'll also make a new controller - **`BLCDataSource`** - which will organize the users, media, and comments, and provide them to our view controller. **`BLCDataSource`** will be the first controller you see in this course that's *not* a view controller.

Since we're not connected to Instragram yet, we'll continue to use placeholder data.

Before we start coding, create a new branch to continue this checkpoint:

Terminal

```
$ git checkout -b refactoring
```

User Object

Create a new Objective-C class in your project. Name it **`BLCUser`** and make it a subclass of **`NSObject`**. Save it in the same directory as all our other classes, which should be the **Blocstagram** directory.

Why **NSObject**?

NSObject is the root class for nearly all Objective-C objects. It gives us basic behavior for allocating and initializing objects. All the classes we create, including **UIViewController**s and **UITableViewCell**s, inherit from this base class.

Open the header file for **BLCUser** and add these properties.

BLCUser.h

```
@interface BLCUser : NSObject

+ @property (nonatomic, strong) NSString *idNumber;
+ @property (nonatomic, strong) NSString *userName;
+ @property (nonatomic, strong) NSString *fullName;
+ @property (nonatomic, strong) NSURL *profilePictureURL;
+ @property (nonatomic, strong) UIImage *profilePicture;

@end
```

We haven't added anything to the implementation; a collection of properties in the interface is enough to create a basic model.

Media Object

Follow the same steps to create the **BLCMedia** class and add the following properties.

BLCMedia.h

```
+ @class BLCUser;

@interface BLCMedia : NSObject

+ @property (nonatomic, strong) NSString *idNumber;
+ @property (nonatomic, strong) BLCUser *user;
+ @property (nonatomic, strong) NSURL *mediaURL;
+ @property (nonatomic, strong) UIImage *image;
+ @property (nonatomic, strong) NSString *caption;
+ @property (nonatomic, strong) NSArray *comments;

@end
```

Note that we use the forward declaration **@class BLCUser;** in this header. You might wonder why we do this instead of just adding **#import BLCUser.h**. We could import it, but it's generally poor practice to import custom classes inside a header file. The reason for this is that we can get into trouble with what's called a **circular inclusion**.

A circular inclusion happens when two classes try to import each other. When this happens we get a compiler error. A circular inclusion can also happen indirectly. For example, **ClassA** imports **ClassB** which imports **ClassC** which imports **ClassD** which imports **ClassA**. This would cause a circular inclusion error.

As we write more classes it can become harder to avoid this problem. We will, however, import the user class in the implementation since we will need it here later.

BLCMedia.m

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

Comment Object:

BLCComment.h

```
+ @class BLCUser;

@interface BLCComment : NSObject

+ @property (nonatomic, strong) NSString *idNumber;

+ @property (nonatomic, strong) BLCUser *from;
+ @property (nonatomic, strong) NSString *text;

@end
```

BLCComment.m

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

Data Source

For the data source we will use something called the **Singleton Pattern**. A singleton is a class that only has one instance. Any code that needs to use this class will share this one instance.

Later, we'll need to access the same data from multiple places in our code base. Without a singleton, we'd either need to:

- write code to pass the data between controllers, or
- refetch the data from the Instagram API

Neither is ideal; we want to have a single source of truth that is shared amongst all of our code.

Create the **`BLCDataSource`** class and define the following method:

BLCDatasource.h

```
@interface BLCDataSource : NSObject

+ +(instancetype) sharedInstance;

@end
```

When we need to access to this class we'll do so by calling **`[BLCDataSource sharedInstance]`**. This will return the single instance that exists for this class. If the instance has not yet been created then this method will create it before returning it. Now let's implement this method:

BLCDataSource.m

```
.

@interface BLCDataSource ()

@property (nonatomic, strong) NSArray *mediaItems;

@end

@implementation BLCDataSource

+ + (instancetype) sharedInstance {
+     static dispatch_once_t once;
+     static id sharedInstance;
+     dispatch_once(&once, ^{
+         sharedInstance = [[self alloc] init];
+     });
+     return sharedInstance;
+ }

@end
```

To make sure we only create a single instance of this class we use a function called **`dispatch_once`**. This function takes a block of code and ensures that it only runs once. We first define a variable to **`static dispatch_once_t once;`**. This variable stores the completion status of the **`dispatch_once`** task.

Next we define a static variable that will hold our shared instance once it's been created: **`static id sharedInstance;`**.

Finally we return the instance: **`return sharedInstance;`**.

Now we'll add a property that we'll use to store our array of **`mediaItems`**.

Open the header file and add this property:

BLCDatasource.h

```
@interface BLCDataSource : NSObject

 + (instancetype) sharedInstance;
 + @property (nonatomic, strong, readonly) NSArray *mediaItems;

@end
```

We'll make this property **`readonly`** to make sure that other classes aren't able to modify it. In the implementation file we will redefine the property without **`readonly`**

BLCDataSource.m

```
#import "BLCDataSource.h"

+ @interface BLCDataSource ()
+
+ @property (nonatomic, strong) NSArray *mediaItems;
+
+ @end

@implementation BLCDataSource

+ (instancetype) sharedInstance {
  static dispatch_once_t once;
  static id sharedInstance;
  dispatch_once(&once, ^{
      sharedInstance = [[self alloc] init];
  });
  return sharedInstance;
}
@end
```

it. Now, we'll write some code to use our existing images, while randomly generating users and comments:

BLCDataSource.m

```
 #import "BLCDataSource.h"
+ #import "BLCUser.h"
+ #import "BLCMedia.h"
+ #import "BLCComment.h"

 @interface BLCDataSource ()

 @property (nonatomic, strong) NSArray *mediaItems;

 @end

 @implementation BLCDataSource

 + (instancetype) sharedInstance {
     static dispatch_once_t once;
     static id sharedInstance;
     dispatch_once(&once, ^{
         sharedInstance = [[self alloc] init];
     });
     return sharedInstance;
 }

+ - (instancetype) init {
+     self = [super init];
+
+     if (self) {
+         [self addRandomData];
+     }
+
+     return self;
+ }

+ - (void) addRandomData {
+     NSMutableArray *randomMediaItems = [NSMutableArray array];
+
+     for (int i = 1; i <= 10; i++) {
+         NSString *imageName = [NSString stringWithFormat:@"%d.jpg", i];
+         UIImage *image = [UIImage imageNamed:imageName];
+
+         if (image) {
+             BLCMedia *media = [[BLCMedia alloc] init];
+             media.user = [self randomUser];
+             media.image = image;
+
+             NSUInteger commentCount = arc4random_uniform(10);
+             NSMutableArray *randomComments = [NSMutableArray array];
+
+             for (int i  = 0; i <= commentCount; i++) {
+                 BLCComment *randomComment = [self randomComment];
+                 [randomComments addObject:randomComment];
+             }
+
+             media.comments = randomComments;
+
+             [randomMediaItems addObject:media];
+         }
+     }
+
+     self.mediaItems = randomMediaItems;
+ }
+
+ - (BLCUser *) randomUser {
+     BLCUser *user = [[BLCUser alloc] init];
+
+     user.userName = [self randomStringOfLength:arc4random_uniform(10)];
+
+     NSString *firstName = [self randomStringOfLength:arc4random_uniform(7)];
```

```
+
+     return user;
+ }
+
+ - (BLCComment *) randomComment {
+     BLCComment *comment = [[BLCComment alloc] init];
+
+     comment.from = [self randomUser];
+
+     NSUInteger wordCount = arc4random_uniform(20);
+
+     NSMutableString *randomSentence = [[NSMutableString alloc] init];
+
+     for (int i  = 0; i <= wordCount; i++) {
+         NSString *randomWord = [self randomStringOfLength:arc4random_uniform(12)];
+         [randomSentence appendFormat:@"%@ ", randomWord];
+     }
+
+     comment.text = randomSentence;
+
+     return comment;
+ }
+
+ - (NSString *) randomStringOfLength:(NSUInteger) len {
+     NSString *alphabet = @"abcdefghijklmnopqrstuvwxyz";
+
+     NSMutableString *s = [NSMutableString string];
+     for (NSUInteger i = 0U; i < len; i++) {
+         u_int32_t r = arc4random_uniform((u_int32_t)[alphabet length]);
+         unichar c = [alphabet characterAtIndex:r];
+         [s appendFormat:@"%C", c];
+     }
+     return [NSString stringWithString:s];
+ }
```

We created a set of methods designed to generate random data for us when the class gets initialized.

This is just placeholder code that will give us some fake data to work with while we build the app. Later, we'll replace this code by hooking into the Instagram API.

There are a few features in our new code we should review:

Random

We use a function called **`arc4random_uniform()`**. This function returns a random number between 0 and the number supplied to it. We use this function to create strings of random length and sentences of random word count.

The **`addRandomData`** method:

- loads every placeholder image in our app
- creates a **`BLCMedia`** model for it
- attaches a randomly generated user to it
- attaches a randomly generated number of comments to it
- puts each media item into the **`mediaItems`** array

Updating our table to use the new data source

The last step of this refactor is to update **`BLCImagesTableViewController`** to use the new data source and models.

First, import all of the new classes and remove the old images property:

BLCImagesTableViewController.m

```
+ #import "BLCDataSource.h"
+ #import "BLCMedia.h"
+ #import "BLCUser.h"
+ #import "BLCComment.h"

@interface BLCImagesTableViewController ()
- @property (nonatomic, strong) NSMutableArray *images;
@end
```

Next, we'll update the **`tableView`** methods to use the new **`BLCDataSource`** singleton:

BLCImagesTableViewController.m

```
- (id)initWithStyle:(UITableViewStyle)style
{
    self = [super initWithStyle:style];
    if (self) {
        // Custom initialization
-         self.images = [NSMutableArray array];
    }
    return self;
}

- (void)viewDidLoad
{
    [super viewDidLoad];

-     for (int i = 1; i <= 10; i++) {
-         NSString *imageName = [NSString stringWithFormat:@"%d.jpg", i];
-         UIImage *image = [UIImage imageNamed:imageName];
-         if (image) {
-             [self.images addObject:image];
-         }
-     }

    [self.tableView registerClass:[UITableViewCell class] forCellReuseIdentifier:@"imageCell"];
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section
{
-     return self.images.count;
+     return [BLCDataSource sharedInstance].mediaItems.count;
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
  static NSInteger imageViewTag = 1234;

  UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:@"imageCell" forIndexPath:indexPath];

  // Configure the cell...

  UIImageView *imageView = (UIImageView*)[cell.contentView viewWithTag:imageViewTag];

  if (!imageView) {
    // This is a new cell, it doesn't have an image view yet
    imageView = [[UIImageView alloc] init];
    imageView.contentMode = UIViewContentModeScaleToFill;

    imageView.frame = cell.contentView.bounds;
    imageView.autoresizingMask = UIViewAutoresizingFlexibleHeight | UIViewAutoresizingFlexibleWidth;

        imageView.tag = imageViewTag;
        [cell.contentView addSubview:imageView];
    }


-     UIImage *image = self.images[indexPath.row];
-     imageView.image = image;
+     BLCMedia *item = [BLCDataSource sharedInstance].mediaItems[indexPath.row];
+     imageView.image = item.image;

    return cell;
}

- (CGFloat) tableView:(UITableView *)tableView heightForRowAtIndexPath:(NSIndexPath *)indexPath {
-     UIImage *image = self.images[indexPath.row];
+     BLCMedia *item = [BLCDataSource sharedInstance].mediaItems[indexPath.row];
+     UIImage *image = item.image;

    return image.size.height / image.size.width * CGRectGetWidth(self.view.frame);
}
```

in fact look and behave the same as before.

In the next checkpoint we'll make the usernames and captions visible.

Notice all the places where we use `[BLCDataSource sharedInstance].mediaItems`. See if you can refactor this code by defining a new method called `items` in `BLCImagesTableViewController.m`. This will simplify our code to read `[self items].count` instead of `[BLCDataSource sharedInstance].mediaItems.count`.

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

Terminal

```
$ git add .
$ git commit -m 'Refactored user names and captions'
$ git checkout master
$ git merge refactoring
$ git push
```

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

assignment completed

MADE BY BLOC

Tech Talks & Resources

Programming Bootcamp Comparison

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

Swiftris: Build Your First iOS Game with Swift

Webflow Tutorial: Design Responsive Sites with Webflow

Ruby Warrior

Bloc's Diversity Scholarship

SIGN UP FOR OUR MAILING LIST

Send

hello@bloc.io

Considering enrolling? (404) 480-2562

Partnership / Corporate Inquiries? (650) 741-5682

Complex Data Cells

Creating A Custom Table Cell

As you can imagine, adding lots of strings and labels to the **`cellForRowAtIndexPath:`** method could make it long and unwieldy. In our **`BlocBrowser`** project, when the toolbar code became too long, we made a **`UIView`** subclass to contain the layout logic.

Similarly, we're going to make a **`UITableViewCell`** subclass to contain our code that displays an Instagram post. Start by creating a new branch for this checkpoint:

Terminal

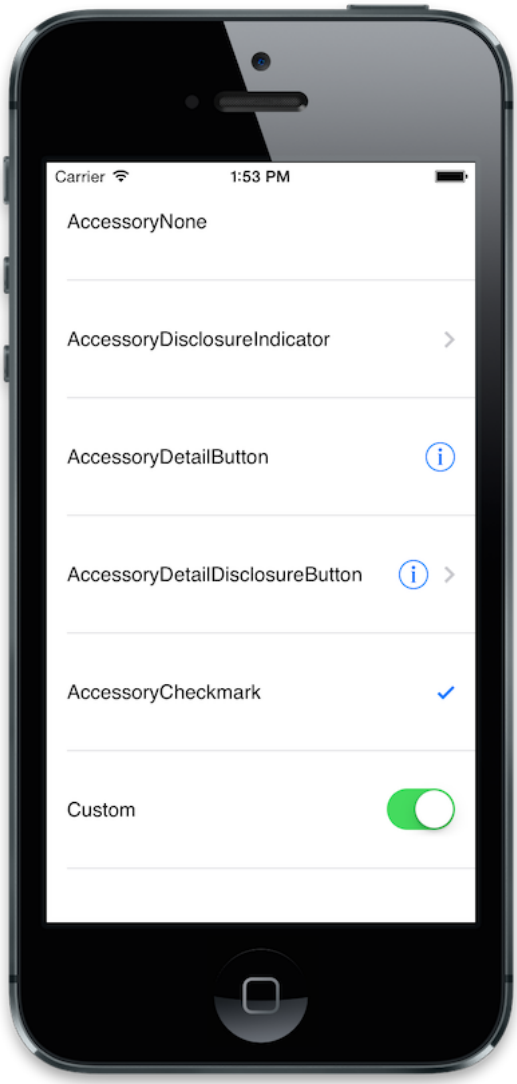
```
$ git checkout -b custom-table-cell
```

Subclassing `UITableViewCell`

Begin by creating a new Objective-C class named **`BLCMediaTableViewCell`** and make sure that its superclass is **`UITableViewCell`**. There are some key differences between subclassing **`UIView`** and **`UITableViewCell`**. In a table cell, we don't add subviews to **`self`**, we add them to **`self.contentView`**. However, the differences don't stop there.

Accessory Views

An **`UITableViewCell`** supports **accessory views**. Accessory views are located on the right edge of an **`UITableViewCell`** and are customizable to be any kind of view you wish. There are five types of accessories built into iOS: **`UITableViewCellAccessoryDisclosureIndicator`**, **`UITableViewCellAccessoryDetailDisclosureButton`**, **`UITableViewCellAccessoryCheckmark`**, **`UITableViewCellAccessoryDetailButton`** and the default, **`UITableViewCellAccessoryNone`**.

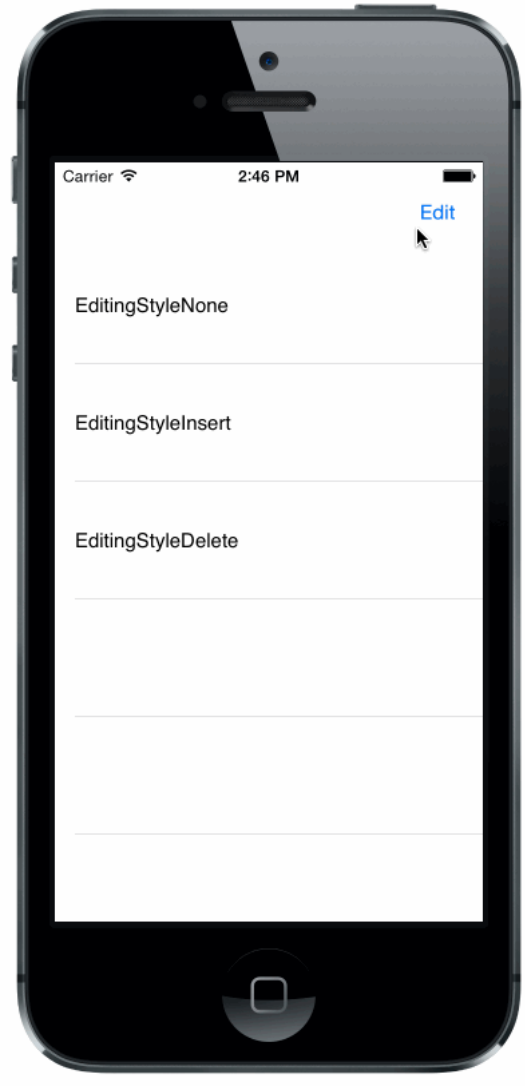


You may have your cell display any one of these five types by setting the **`accessoryStyle`** property to the respective value. To provide your own, set **`accessoryStyle`** to **`UITableViewCellAccessoryNone`** and the **`accessoryView`** property to any view you've created. In the screenshot, the view used in the custom cell is an **`UISwitch`**.

Editing Mode

A table view may be entered into an "editing mode" by invoking **`setEditing:animated:`** on **`UITableView`**. Editing views are found on the left edge of the table cell. There are three iOS-provided styles for editing: **`UITableViewCellEditingStyleDelete`**, **`UITableViewCellEditingStyleInsert`** and the default, **`UITableViewCellEditingStyleNone`**.

this method, the specific cell at that index path will be editable when that table view is placed into editing mode:



Interface

Let's begin, as always, with the interface:

BLCMediaTableViewCell.h

```
+ @class BLCMedia;
@interface BLCMediaTableViewCell : UITableViewCell
+ @property (nonatomic, strong) BLCMedia *mediaItem;
@end
```

Each cell will be associated with a single media item. When we use this cell later in our **`BLCImagesTableViewController`**, we'll only have to update this property in order to customize the cell for display.

Implementation

Begin by adding some imports. We're going to need access to these classes:

BLCMediaTableViewCell.m

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

Our custom views will be set up as properties:

BLCMediaTableViewCell.m

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

Initialize them and add them to **`self.contentView`**:

BLCMediaTableViewCell.m

```
- (id)initWithStyle:(UITableViewCellStyle)style reuseIdentifier:(NSString *)reuseIdentifier
{
    self = [super initWithStyle:style reuseIdentifier:reuseIdentifier];
    if (self) {
        // Initialization code
+         self.mediaImageView = [[UIImageView alloc] init];
+         self.usernameAndCaptionLabel = [[UILabel alloc] init];
+         self.usernameAndCaptionLabel.numberOfLines = 0;
+         self.commentLabel = [[UILabel alloc] init];
+         self.commentLabel.numberOfLines = 0;

+         for (UIView *view in @[self.mediaImageView, self.usernameAndCaptionLabel, self.commentLabel]) {
+             [self.contentView addSubview:view];
+         }
    }
    return self;
}
```

This code should look self-explanatory and familiar. Let's dig into new stuff, *attributed strings.*

Styling Our Cell

Attributed strings are not string objects. In fact, they aren't a subclass of **`NSString`** at all. An **`NSAttributedString`** is more than a collection of characters, it also provides appearance attributes for those characters. If you've ever worked with a word processor, perhaps one whose name rhymes with *LauraCroft Bird*, you've probably *italicized*, **bolded**, applied a custom font, colored or indented portions of your document.

Attributed strings let us perform these **rich-text modifications** on strings for presentation in iOS. We want our usernames and comments t resemble those found on Instagram and we'll accomplish this using attributed strings.

Let's begin with preparing our attributes. Add the following **`static`** variables to our cell:

BLCMediaTableViewCell.m

```
@interface BLCMediaTableViewCell ()
@property (nonatomic, strong) UIImageView *mediaImageView;
@property (nonatomic, strong) UILabel *usernameAndCaptionLabel;
@property (nonatomic, strong) UILabel *commentLabel;
@end

+ static UIFont *lightFont;
+ static UIFont *boldFont;
+ static UIColor *usernameLabelGray;
+ static UIColor *commentLabelGray;
+ static UIColor *linkColor;
+ static NSParagraphStyle *paragraphStyle;

@implementation BLCMediaTableViewCell
```

color for the username and caption label whereas the **commentLabelGray** will be a separate background color for the comment section.

linkColor will be the text color of every username in order to make it appear tap-able. Finally, an **NSParagraphStyle** lets us set properties lik line spacing, text alignment, indentation, paragraph spacing, etc.

load

Since the variables we've declared were all **static** and therefore belong to every instance of **BLCMediaTableViewCell**, we're going to initialize them in a method named **load**. **load** is a special method which is called *once and only once* per class. Any class may choose to implement **loa** If it does, when the class is first used, the method will be executed before anything else happens:

BLCMediaTableViewCell.m

```
+ + (void)load {
+     lightFont = [UIFont fontWithName:@"HelveticaNeue-Thin" size:11];
+     boldFont = [UIFont fontWithName:@"HelveticaNeue-Bold" size:11];
+     usernameLabelGray = [UIColor colorWithRed:0.933 green:0.933 blue:0.933 alpha:1]; /*#eeeeee*/
+     commentLabelGray = [UIColor colorWithRed:0.898 green:0.898 blue:0.898 alpha:1]; /*#e5e5e5*/
+     linkColor = [UIColor colorWithRed:0.345 green:0.314 blue:0.427 alpha:1]; /*#58506d*/

+     NSMutableParagraphStyle *mutableParagraphStyle = [[NSMutableParagraphStyle alloc] init];
+     mutableParagraphStyle.headIndent = 20.0;
+     mutableParagraphStyle.firstLineHeadIndent = 20.0;
+     mutableParagraphStyle.tailIndent = -20.0;
+     mutableParagraphStyle.paragraphSpacingBefore = 5;

+     paragraphStyle = mutableParagraphStyle;
}
```

The fonts we've chosen come pre-packaged with iOS (a full list is **available here**). The colors are a couple shades of gray and **linkColor** is a shade of purple. Lastly, the paragraph style is established. The first line of our paragraphs will be indented by **20** points as well as all subsequent lines.

The tail indent specifies where the ends of the lines should stop. Given a positive value, it marks the distance in points from the *left-most* edge given a negative value like **-20**, it provides the distance from the *right-most* edge. Lastly, we set **paragraphSpacingBefore** which indicates how far each new paragraph should be from the previous, **5** points in our case.

Attributing Our Strings

Let's create our first attributed string. We'll do so from a method titled **usernameAndCaptionString**:

BLCMediaTableViewCell.m

```
+ - (NSAttributedString *) usernameAndCaptionString {
+     CGFloat usernameFontSize = 15;

+     // Make a string that says "username caption text"
+     NSString *baseString = [NSString stringWithFormat:@"%@ %@", self.mediaItem.user.userName, self.mediaItem.caption];

+     // Make an attributed string, with the "username" bold
+     NSMutableAttributedString *mutableUsernameAndCaptionString = [[NSMutableAttributedString alloc] initWithString:baseString attri

+     NSRange usernameRange = [baseString rangeOfString:self.mediaItem.user.userName];
+     [mutableUsernameAndCaptionString addAttribute:NSFontAttributeName value:[boldFont fontWithSize:usernameFontSize] range:username
+     [mutableUsernameAndCaptionString addAttribute:NSForegroundColorAttributeName value:linkColor range:usernameRange];

+     return mutableUsernameAndCaptionString;
}
```

Let's break down what's happening here. First we choose a font size, **15**. It will be the consistent font size for both the username and the caption. Then we create a base string which just ends up looking like, "username caption". It's a simple concatenation of the username and caption with a space in between.

as an **NSDictionary** which provides the font we'd like to use as well as the paragraph style. The attributes in the dictionary will apply to the *entire* attributed string.

However, we only want the username to be **bold** and purple colored, therefore we recover the **NSRange** of the username within the base strir and apply our bold font to it and then our link color. These attributes *override* the ones which we set previously using the dictionary but only for the **NSRange** we specify.

We're going to have a separate method for generating our comment attributed string:

BLCMediaTableViewCell.m

```
+ - (NSAttributedString *) commentString {
+     NSMutableAttributedString *commentString = [[NSMutableAttributedString alloc] init];

+     for (BLCComment *comment in self.mediaItem.comments) {
+         // Make a string that says "username comment text" followed by a line break
+         NSString *baseString = [NSString stringWithFormat:@"%@ %@\n", comment.from.userName, comment.text];

+         // Make an attributed string, with the "username" bold

+         NSMutableAttributedString *oneCommentString = [[NSMutableAttributedString alloc] initWithString:baseString attributes:@{NSF

+         NSRange usernameRange = [baseString rangeOfString:comment.from.userName];
+         [oneCommentString addAttribute:NSFontAttributeName value:boldFont range:usernameRange];
+         [oneCommentString addAttribute:NSForegroundColorAttributeName value:linkColor range:usernameRange];

+         [commentString appendAttributedString:oneCommentString];
+     }

+     return commentString;
+ }
```

We perform a very similar set of instructions in this method. However, **commentString** is actually a concatenation of every comment found fo that particular media item. We use a **for** loop to iterate over each comment and create its own respective attributed string which we then append to **commentString** once it's ready.

Layout Subviews

Just like subclassing **UIView** to build our custom toolbar, we need to implement **layoutSubviews**. First, we're going to write a method which helps us calculate the size of our attributed strings:

BLCMediaTableViewCell.m

```
+ - (CGSize) sizeOfString:(NSAttributedString *)string {
+     CGSize maxSize = CGSizeMake(CGRectGetWidth(self.contentView.bounds) - 40, 0.0);
+     CGRect sizeRect = [string boundingRectWithSize:maxSize options:NSStringDrawingUsesLineFragmentOrigin context:nil];
+     sizeRect.size.height += 20;
+     sizeRect = CGRectIntegral(sizeRect);
+     return sizeRect.size;
+ }
```

The purpose of this method is for us to easily calculate how tall our **usernameAndCaptionLabel** and **commentLabel**s need to be. The **boundingRectWithSize:options:context:** method will take the text, the attributes and the maximum width we've supplied, **280**, to determine how much space our string requires.

We do this so that later when we layout our views, we don't cut-off any of the text nor do we give it more room than it needs. We also add a height of **20** just to pad out the top and bottom which gives the text some breathing room. And speaking of laying out views:

BLCMediaTableViewCell.m

```
+     [super layoutSubviews];

+     CGFloat imageHeight = self.mediaItem.image.size.height / self.mediaItem.image.size.width * CGRectGetWidth(self.contentView.boun
+     self.mediaImageView.frame = CGRectMake(0, 0, CGRectGetWidth(self.contentView.bounds), imageHeight);

+     CGSize sizeOfUsernameAndCaptionLabel = [self sizeOfString:self.usernameAndCaptionLabel.attributedText];
+     self.usernameAndCaptionLabel.frame = CGRectMake(0, CGRectGetMaxY(self.mediaImageView.frame), CGRectGetWidth(self.contentView.bo

+     CGSize sizeOfCommentLabel = [self sizeOfString:self.commentLabel.attributedText];
+     self.commentLabel.frame = CGRectMake(0, CGRectGetMaxY(self.usernameAndCaptionLabel.frame), CGRectGetWidth(self.bounds), sizeOfC

+     // Hide the line between cells
+     self.separatorInset = UIEdgeInsetsMake(0, 0, 0, CGRectGetWidth(self.bounds));
+ }
```

We begin by re-using the code found originally in **`[BLCImagesTableViewController -tableView:heightForRowAtIndexPath:]`** that calculates the size of the image. The rest is pretty standard: we call our **`sizeOfString:`** method to get the height for the username / caption label and then frame it right beneath the image.

We follow that up by repeating that logic for the comment string and place it beneath the username / caption label. Lastly, we perform a trick which will hide the divider line typically seen between table cells. We don't need it because the next image will be a clear indicator that a new cell has started.

Setting The Media Item

The last but certainly not least important piece is setting the media item for the cell. When we declare a property like the one in our header file, **`@property (nonatomic, strong) BLCMedia *mediaItem;`** the compiler generates two hidden methods for us, a *getter* and a *setter*:

BLCMediaTableViewCell.h

```
// Get the media item
- (BLCMedia *)mediaItem;
// Set a new media item
- (void)setMediaItem:(BLCMedia *)mediaItem;
```

When working with the **`mediaItem`** property, assignments like these:

```
mediaItemCell.mediaItem = myNewMediaItem;
```

Are just syntactical shortcuts for this:

```
[mediaItemCell setMediaItem:myNewMediaItem];
```

Likewise, when reading the property:

```
BLCMedia *item = mediaItemCell.mediaItem;
```

You're actually calling the auto-generated getter:

```
BLCMedia *item = [mediaItemCell mediaItem];
```

What's nifty about these auto-generated methods is that you can actually supply your own in order to define custom behaviors. What we war to do is update the image and text labels whenever a new media item is set. We're going to accomplish this by providing our own setter method:

BLCMediaTableViewCell.m

```
+      _mediaItem = mediaItem;
+      self.mediaImageView.image = _mediaItem.image;
+      self.usernameAndCaptionLabel.attributedText = [self usernameAndCaptionString];
+      self.commentLabel.attributedText = [self commentString];
+  }
```

_mediaItem is an auto-generated instance variable. Imagine it being created like so, but implicitly for you during compilation:

BLCMediaTableViewCell.m

```
@interface BLCMediaTableViewCell () {
    BLCMedia* _mediaItem;
}
```

When overriding a setter or getter method for a property, you must refer to the implicitly generated IVAR (instance-variable) rather than the property itself. The IVAR will always be named **_{propertyName}**. Referring to the property will cause an infinite loop. For instance, if we changed **setMediaItem:** to the following:

```
- (void) setMediaItem:(BLCMedia *)mediaItem {
    self.mediaItem = mediaItem; // Causes an infinite recursion
    self.mediaImageView.image = _mediaItem.image;
    self.usernameAndCaptionLabel.attributedText = [self usernameAndCaptionString];
    self.commentLabel.attributedText = [self commentString];
}
```

The very first line in this method, **self.mediaItem = mediaItem;** will cause an infinite recursion because it is invoking **[self setMediaItem:mediaItem]**. Remember that **self.property = variable;** is just shorthand for **[self setProperty:variable]**.

Updating Our Table View

All that's left is to update our table view controller. Let's begin by importing our hot new custom table cell:

BLCImagesTableViewController.m

```
#import "BLCImagesTableViewController.h"
#import "BLCDataSource.h"
#import "BLCMedia.h"
#import "BLCUser.h"
#import "BLCComment.h"
+ #import "BLCMediaTableViewCell.h"
```

Next, we need to update the class which we register with our table view, it's no longer the generic **UITableViewCell** anymore:

BLCImagesTableViewController.m

```
-      [self.tableView registerClass:[UITableViewCell class] forCellReuseIdentifier:@"imageCell"];
+      [self.tableView registerClass:[BLCMediaTableViewCell class] forCellReuseIdentifier:@"mediaCell"];
```

Let's modify the **tableView:heightForRowAtIndexPath:** method. It's not an exact approach and we'll see why shortly but it'll do for now:

BLCImagesTableViewController.m

```
- (CGFloat) tableView:(UITableView *)tableView heightForRowAtIndexPath:(NSIndexPath *)indexPath {
    BLCMedia *item = [BLCDataSource sharedInstance].mediaItems[indexPath.row];
    UIImage *image = item.image;
-      return image.size.height / image.size.width * CGRectGetWidth(self.view.frame);
+      return 300 + (image.size.height / image.size.width * CGRectGetWidth(self.view.frame));
}
```

Most impressively, let's cut our **tableView:cellForRowAtIndexPath:** method down to size:

```
-      static NSInteger imageViewTag = 1234;
-
-      UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:@"imageCell" forIndexPath:indexPath];
-
-      // Configure the cell...
-
-      UIImageView *imageView = (UIImageView*)[cell.contentView viewWithTag:imageViewTag];

-      if (!imageView) {
-          // This is a new cell, it doesn't have an image view yet
-          imageView = [[UIImageView alloc] init];
-          imageView.contentMode = UIViewContentModeScaleToFill;

-          imageView.frame = cell.contentView.bounds;
-          imageView.autoresizingMask = UIViewAutoresizingFlexibleHeight | UIViewAutoresizingFlexibleWidth;

-          imageView.tag = imageViewTag;
-          [cell.contentView addSubview:imageView];
-      }
-
-      BLCMedia *item = [BLCDataSource sharedInstance].mediaItems[indexPath.row];
-      imageView.image = item.image;
-
+      BLCMediaTableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:@"mediaCell" forIndexPath:indexPath];
+      cell.mediaItem = [BLCDataSource sharedInstance].mediaItems[indexPath.row];
    return cell;
}
```

Two lines, *two*! Run the application and you should see something resembling the following:

As you can see, there's extra space in some rows. They didn't need the entirety of the height we supplied in **`tableView:heightForRowAtIndexPath:`** which is our original image height calculation plus **`300`** points, a random choice. We need to be able t specify the precise height for each cell and since no two cells are identical, we're going to have to add another method to **`BLCMediaTableViewCell`**.

Height Calculation

Let's add a method to the interface of our table cell:

BLCMediaTableViewCell.h

```
@class BLCMedia;
@interface BLCMediaTableViewCell : UITableViewCell

@property (nonatomic, strong) BLCMedia *mediaItem;

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

@end
```

Instead of a **-** before the method signature, we've placed a **+**. The **+** signifies that this method does not belong to an instance of that object, it belongs to the class. It's kind of like the **`static`** variables we declared earlier which belong to all instances except we've declared it in the header so that any other class may use it. You don't need to allocate a copy of **`BLCMediaTableViewCell`** in order to call this method, you can invoke it like so:

```
[BLCMediaTableViewCell heightForMediaItem:mediaItem width:320];
```

> *Notice that the target of the method is the full class name,* **`BLCMediaTableViewCell`**.

This method will be responsible for "faking" a layout event such that we can get the full height of a completed cell as if it were actually being placed into a table:

BLCMediaTableViewCell.m

```
+ (CGFloat) heightForMediaItem:(BLCMedia *)mediaItem width:(CGFloat)width {
+      // Make a cell
+      BLCMediaTableViewCell *layoutCell = [[BLCMediaTableViewCell alloc] initWithStyle:UITableViewCellStyleDefault reuseIdentifier:@"

+      // Set it to the given width, and the maximum possible height
+      layoutCell.frame = CGRectMake(0, 0, width, CGFLOAT_MAX);

+      // Give it the media item
+      layoutCell.mediaItem = mediaItem;

+      // Make it adjust the image view and labels
+      [layoutCell layoutSubviews];

+      // The height will be wherever the bottom of the comments label is
+      return CGRectGetMaxY(layoutCell.commentLabel.frame);
}
```

The method is fairly self-explanatory. We create a local copy of the cell which will mirror the one actually being used in the table view controller. We call **`layoutSubviews`** on it and once that method has completed, it should be appropriately sized to fit all of its contents. We return the height of our temporary dummy cell.

Let's update our table controller to make use of this new method:

BLCImagesTableViewController.m

```
- (CGFloat) tableView:(UITableView *)tableView heightForRowAtIndexPath:(NSIndexPath *)indexPath {
    BLCMedia *item = [BLCDataSource sharedInstance].mediaItems[indexPath.row];
-      UIImage *image = item.image;
-      return 300 + image.size.height / image.size.width * CGRectGetWidth(self.view.frame);
+      return [BLCMediaTableViewCell heightForMediaItem:item width:CGRectGetWidth(self.view.frame)];
}
```

Run the app and rejoice!

- Modify **`BLCImagesTableViewController`** to support editing mode.
- Experiment with deleting rows.

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

Terminal

```
$ git add .
$ git commit -m 'Switched to a custom table view cell'
$ git checkout master
$ git merge custom-table-cell
$ git push
```

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

assignment completed

Roadmap · Previous Checkpoint Jump to Submission · Next Checkpoir

Manageable Layouts

Our table cell content has three views: an image view and two labels. Currently, laying them out requires us to set frames in **`layoutSubviews`** and calculate string size in **`sizeOfString:`**.

This is called *manual layout* or *frame-based layout*.

We can alternatively use *auto-layout*, or *constraint-based layout*.

Auto layout is a system for laying out an interface by describing the relationships between views.

This has the advantage of easier-to-read code, and less manual math (though it can be **less performant**). Despite its drawbacks, Apple heavily encourages the use of auto-layout for all future iOS development, since it can more simply handle different screen sizes, device orientations, and aspect ratios.

Let's explore how auto layout will work for our cell, and then we'll go over some more details.

Create a new branch:

Terminal

```
$ git checkout -b auto-layout-cell
```

Converting our Cell to Auto-Layout

We won't be needing **`sizeOfString:`** any more. Delete it.

BLCMediaTableViewCell.m

```
- - (CGSize) sizeOfString:(NSAttributedString *)string {
-     CGSize maxSize = CGSizeMake(CGRectGetWidth(self.contentView.bounds) - 20, 0.0);
-     CGRect sizeRect = [string boundingRectWithSize:maxSize options:NSStringDrawingUsesLineFragmentOrigin context:nil];
-     sizeRect.size.height+= 20;
-     sizeRect = CGRectIntegral(sizeRect);
-     return sizeRect.size;
- }
```

Instead of **setting frames**, we're going to be **adding constraints** to our views. For a **`UIView`** subclass, the **`init`** method is a good place to add them.

Here are some basics:

- A layout constraint is represented by the **`NSLayoutConstraint`** class.
- For auto-layout to work, you must provide enough constraints to unambiguously determine both a view's size and position.
- Most of the time, each view will get four constraints: height, width, a top constraint, and a left (or "leading") constraint.
- Some views, like **`UILabel`** and **`UIButton`**, have an *intrinsic* content size - that is, a size they *want* to be. This means you can optionally ignore the height and width constraints for these classes.

Examples should help. Let's add some properties for some of our constraints:

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

You don't need a property for every constraint, just the ones you'll want to change later.

In **`init`**, we need to make one small change before we start adding constraints:

BLCMediaTableViewCell.m

```
for (UIView *view in @[self.mediaImageView, self.usernameAndCaptionLabel, self.commentLabel]) {
    [self.contentView addSubview:view];
+     view.translatesAutoresizingMaskIntoConstraints = NO;
}
```

`translatesAutoresizingMaskIntoConstraints` converts the auto-resizing mask you learned about earlier into constraints automatically. Usually, this is set to **`NO`** when working with auto-layout.

Now let's add some constraints. The first ones we'll add using a "visual formatting string", which lets you draw a rough outline of your views using only keyboard characters:

BLCMediaTableViewCell.m

```
for (UIView *view in @[self.mediaImageView, self.usernameAndCaptionLabel, self.commentLabel]) {
    [self.contentView addSubview:view];
    view.translatesAutoresizingMaskIntoConstraints = NO;
}

+ NSDictionary *viewDictionary = NSDictionaryOfVariableBindings(_mediaImageView, _usernameAndCaptionLabel, _commentLabel);
+
+ [self.contentView addConstraints:[NSLayoutConstraint constraintsWithVisualFormat:@"H:|[_mediaImageView]|" options:kNilOptions metri
+ [self.contentView addConstraints:[NSLayoutConstraint constraintsWithVisualFormat:@"H:|[_usernameAndCaptionLabel]|" options:kNilOpti
+ [self.contentView addConstraints:[NSLayoutConstraint constraintsWithVisualFormat:@"H:|[_commentLabel]|" options:kNilOptions metrics
+
+ [self.contentView addConstraints:[NSLayoutConstraint constraintsWithVisualFormat:@"V:|[_mediaImageView][_usernameAndCaptionLabel][_
+                                                                           options:kNilOptions
+                                                                           metrics:nil
+                                                                             views:viewDictionary]];
```

Each visual format string should begin with **`H:`** (horizontal) or **`V:`** (vertical). **`|`** represents the superview, and **`[someName]`** represents one view.

- **`H:|[_mediaImageView]|`** means **`_mediaImageView`** should exactly match the width of its superview.
- **`H:|[_usernameAndCaptionLabel]|`** and **`H:|[_commentLabel]|`** mean the same for the two text labels.
- **`V:|[_mediaImageView][_usernameAndCaptionLabel][_commentLabel]`** means that the three views should be stacked vertically, from the top, with no space in between.

In these three lines, we've added constraints that unambiguously specify the width, top, and left values for all three views.

The only constraints remaining are the height constraints. Let's add these without the visual format string:

BLCMediaTableViewCell.m

```
+                                                    attribute:NSLayoutAttributeHeight
+                                                    relatedBy:NSLayoutRelationEqual
+                                                       toItem:nil
+                                                    attribute:NSLayoutAttributeNotAnAttribute
+                                                   multiplier:1
+                                                     constant:100];
+
+
+  self.usernameAndCaptionLabelHeightConstraint = [NSLayoutConstraint constraintWithItem:_usernameAndCaptionLabel
+                                                                               attribute:NSLayoutAttributeHeight
+                                                                               relatedBy:NSLayoutRelationEqual
+                                                                                  toItem:nil
+                                                                               attribute:NSLayoutAttributeNotAnAttribute
+                                                                              multiplier:1
+                                                                                constant:100];
+
+  self.commentLabelHeightConstraint = [NSLayoutConstraint constraintWithItem:_commentLabel
+                                                                    attribute:NSLayoutAttributeHeight
+                                                                    relatedBy:NSLayoutRelationEqual
+                                                                       toItem:nil
+                                                                    attribute:NSLayoutAttributeNotAnAttribute
+                                                                   multiplier:1
+                                                                     constant:100];

+  [self.contentView addConstraints:@[self.imageHeightConstraint, self.usernameAndCaptionLabelHeightConstraint, self.commentLabelHeig
```

In all three of these constraints, we're saying that the height of the view should equal 100 points. (We'll update that number later, once the content is set.)

Here's a bit about what's going on:

This method has many parameters, because it's very flexible. To help you think about it, try translating it into English. For example, this:

```
[NSLayoutConstraint constraintWithItem:_commentLabel
                             attribute:NSLayoutAttributeHeight
                             relatedBy:NSLayoutRelationEqual
                                toItem:nil
                             attribute:NSLayoutAttributeNotAnAttribute
                            multiplier:1
                              constant:100];
```

Means "the **height** of **`_commentLabel`** is **equal** to (*nothing* * 1) + 100."

This method can create many different constraints. Here are some examples:

- The **width** of **`mcDonaldsLabel`** is **equal** to (the **width** of **`goldenArchesImageView`** * 1) + 0
- The **height** of **`squareButton`** is **equal** to (the **height** of **`squareButton`** * 1) + 0
- The **width** of **`moviePlayer`** is **greater than or equal to** (the **height** of **`moviePlayer`** * 1.77) + 0

For more info, see Apple's **Working With Auto Layout Guide**.

We'll update **`layoutSubviews:`** and **`setMediaItem:`** to provide the correct heights:

BLCMediaTableViewCell.m

```
    [super layoutSubviews];

-       CGFloat imageHeight = self.mediaItem.image.size.height / self.mediaItem.image.size.width * CGRectGetWidth(self.contentView.bou
-       self.mediaImageView.frame = CGRectMake(0, 0, CGRectGetWidth(self.contentView.bounds), imageHeight);
-
-       CGSize sizeOfUsernameAndCaptionLabel = [self sizeOfString:self.usernameAndCaptionLabel.attributedText];
-       self.usernameAndCaptionLabel.frame = CGRectMake(0, CGRectGetMaxY(self.mediaImageView.frame), CGRectGetWidth(self.contentView.b
-
-       CGSize sizeOfCommentLabel = [self sizeOfString:self.commentLabel.attributedText];
-       self.commentLabel.frame = CGRectMake(0, CGRectGetMaxY(self.usernameAndCaptionLabel.frame), CGRectGetWidth(self.bounds), sizeOf
+       // Before layout, calculate the intrinsic size of the labels (the size they "want" to be), and add 20 to the height for some v
+       CGSize maxSize = CGSizeMake(CGRectGetWidth(self.bounds), CGFLOAT_MAX);
+       CGSize usernameLabelSize = [self.usernameAndCaptionLabel sizeThatFits:maxSize];
+       CGSize commentLabelSize = [self.commentLabel sizeThatFits:maxSize];

+       self.usernameAndCaptionLabelHeightConstraint.constant = usernameLabelSize.height + 20;
+       self.commentLabelHeightConstraint.constant = commentLabelSize.height + 20;
+
    // Hide the line between cells
    self.separatorInset = UIEdgeInsetsMake(0, 0, 0, CGRectGetWidth(self.bounds));
}

- (void) setMediaItem:(BLCMedia *)mediaItem {
    _mediaItem = mediaItem;
    self.mediaImageView.image = _mediaItem.image;
    self.usernameAndCaptionLabel.attributedText = [self usernameAndCaptionString];
    self.commentLabel.attributedText = [self commentString];
+
+       self.imageHeightConstraint.constant = self.mediaItem.image.size.height / self.mediaItem.image.size.width * CGRectGetWidth(self
}
```

In **`setMediaItem:`**, we're using the same image calculation to determine the appropriate height of the image.

In **`layoutSubviews`**, we ask the labels for their intrinsic size, add some padding, and set the height constraint constant to that number. (This overwrites the previous **`100`** value.)

If we didn't add the height constraints to the labels, the intrinsic height without the padding would automatically be used instead.

Finally, we need to make slight updates to **`heightForMediaItem:width:`**:

BLCMediaTableViewCell.m

```
+ (CGFloat) heightForMediaItem:(BLCMedia *)mediaItem width:(CGFloat)width {
    // Make a cell
    BLCMediaTableViewCell *layoutCell = [[BLCMediaTableViewCell alloc] initWithStyle:UITableViewCellStyleDefault reuseIdentifier:@"l
-
-       // Set it to the given width, and the maximum possible height
-       layoutCell.frame = CGRectMake(0, 0, width, CGFLOAT_MAX);
-
-       // Give it the media item
    layoutCell.mediaItem = mediaItem;

-       // Make it adjust the image view and labels
-       [layoutCell layoutSubviews];
+       layoutCell.frame = CGRectMake(0, 0, width, CGRectGetHeight(layoutCell.frame));

-       // The height will be wherever the bottom of the comments label is
+       [layoutCell setNeedsLayout];
+       [layoutCell layoutIfNeeded];
+
+       // Get the actual height required for the cell
    return CGRectGetMaxY(layoutCell.commentLabel.frame);
}
```

Run the app and it should work the same. Less code, less math, and the same result.

For some views, auto layout can save you hundreds of lines of code. It's also helpful for easier iPad compatibility, and future-proofing against

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 auto layout'
$ git checkout master
$ git merge auto-layout-cell
$ git push
```

Play with auto-layout on a separate branch:

Terminal

```
$ git checkout -b auto-layout-cell-assignment
```

- Use auto layout to distribute the three views horizontally instead of vertically.
- Use auto layout to place the text on the image (like a magazine cover)

Commit your assignment, and push this separate branch to GitHub *without* merging it into master. Then check out master again:

Terminal

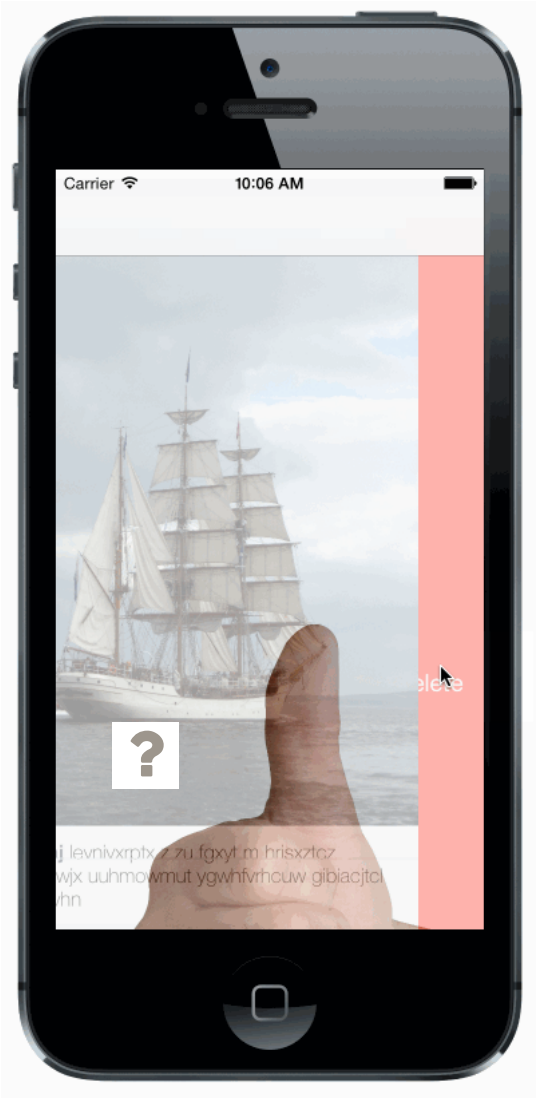
```
$ git commit -am "Auto layout assignment"
$ git push -u origin auto-layout-cell-assignment
$ git checkout master
```

Send a message to your mentor with a link to your assignment branch on GitHub.

You may want to review **Auto Layout Concepts** and the **Visual Formatting Language**. You don't need to memorize this material, but it's good to have read over it once.

assignment completed

Deleting Images



> *"Use a picture. It's worth a thousand words."*
>
> *- Newspaper editor Arthur Brisbane, Syracuse Post Standard (page 18). March 28, 1911*

Deleting Images

Pictures are wonderful to look at... except when they're awful. Sometimes, you just don't want to see what someone else posted to Instagram In this checkpoint, we'll add the ability to swipe-to-delete photos to our Instagram app.

This will require a few steps:

2. We need to enable swipe-to-delete on the table view
3. When a user swipes to delete, the table view delegate (our view controller) needs to inform our data source to remove the image
4. When an image is deleted, our data source needs to inform our view controller that an image was deleted
5. When our view controller is informed that an image was deleted, it needs to tell the table view to hide that cell

There are two different approaches we could go with here:

- We could write a bunch of delegate methods to send messages back and forth (like **`deleteMediaItem:`**, and then a response like **`dataSource:successfullyDeletedMediaItemAtIndex:`**)
- We could use a new way to communicate between objects called *Key-Value Observing* that does a lot of the work for us.

The second approach will teach you something new, and we'll end up with less code to write. Let's do that!

First, create a new branch:

Terminal

```
$ git checkout -b key-value-observation
```

About Key-Value Observing

In its simplest form, key-value observing allows one object to be notified when another object's property changes. In our case, **`BLCImagesTableViewController`** will register to be notified when **`BLCDataSource`**'s **`mediaItems`** property changes, **`mediaItems`** being the key.

In order for us to accomplish that, we must first make our **`mediaItems`** **key-value compliant**, or KVC. Key-value compliance requires that **`mediaItems`** be accessible in a particular way. In an array's case, we need to add several methods and modifications before other classes may register for key-value observation, or KVO:

BLCDataSource.m

```
- @interface BLCDataSource ()
+ @interface BLCDataSource () {
+     NSMutableArray *_mediaItems;
+ }
```

This first step is a requirement of KVC. An array must be accessible as an instance variable named **`_<key>`** or by a method named **`-<key>`** which returns a reference to the array; we're opting for the former. Next, we're going to add accessor methods which will allow observers to be notified when the content of the array changes:

BLCDataSource.m

```
+ #pragma mark - Key/Value Observing

+ - (NSUInteger) countOfMediaItems {
+     return self.mediaItems.count;
+ }

+ - (id) objectInMediaItemsAtIndex:(NSUInteger)index {
+     return [self.mediaItems objectAtIndex:index];
+ }

+ - (NSArray *) mediaItemsAtIndexes:(NSIndexSet *)indexes {
+     return [self.mediaItems objectsAtIndexes:indexes];
+ }
```

These methods will be discovered by key name. For instance, **`countOf<Key>`** is one of the required accessor methods. It indicates that the signature must begin with **`countOf`** which must then be proceeded by the capitalized name of your key. Our key is **`mediaItems`** and the capitalized version of it is **`MediaItems`**, hence the full signature of the method will be **`countOfMediaItems`**.

Next, we'll need to add mutable accessor methods. These are KVC methods which allow insertion and deletion of elements from **`mediaItems`**:

```
+ - (void) insertObject:(BLCMedia *)object inMediaItemsAtIndex:(NSUInteger)index {
+     [_mediaItems insertObject:object atIndex:index];
+ }

+ - (void) removeObjectFromMediaItemsAtIndex:(NSUInteger)index {
+     [_mediaItems removeObjectAtIndex:index];
+ }

+ - (void) replaceObjectInMediaItemsAtIndex:(NSUInteger)index withObject:(id)object {
+     [_mediaItems replaceObjectAtIndex:index withObject:object];
+ }
```

As you can see, this code is quite simple. The methods map almost directly to those found natively within **`NSMutableArray`**. In these three methods, however, note that we reference our array using its instance variable declaration, **`_mediaItems`**. Remember, this is done because in our header file, **`mediaItems`** is declared as **`readonly`** whereas in our implementation file, **`_mediaItems`** is modifiable. If we attempted to write these methods using **`self.mediaItems`**, Xcode would mark those lines as syntax errors, **`attempting to write to a readonly property.`**

And that's it! Our array of items is now key-value compliant and we can proceed to the next step, registering for observation.

Registering for Key-Value Notifications

Registering for KVO is a little bit clunky. Thankfully, it's simplified for our purposes since **`BLCImagesTableViewController`** will only observe a single key. Let's begin by registering for KVO of **`mediaItems`** inside of our **`viewDidLoad`** method:

BLCImagesTableViewController.m

```
- (void)viewDidLoad
{
    [super viewDidLoad];

+     [[BLCDataSource sharedInstance] addObserver:self forKeyPath:@"mediaItems" options:0 context:nil];

    [self.tableView registerClass:[BLCMediaTableViewCell class] forCellReuseIdentifier:@"mediaCell"];
}
```

Whenever a class is added as an observer, it must also remove itself as an observer later. Let's implement the **`dealloc`** method for **`BLCImagesTableViewController`** and stop observing **`mediaItems`**:

BLCImagesTableViewController.m

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

`dealloc` is a method found in **`NSObject`** and its purpose is to allow an instance of your class to perform some last-second cleanup before it goes away. We discussed memory management as well as ARC in an earlier checkpoint. Specifically, you learned that once an instance no longer has any **`strong`** references to it, it will be *deallocated* and the memory it occupied will be reclaimed by the operating system.

`dealloc` is the opposite of **`alloc`**: instead of requesting the required space in memory, the memory is being taken away. When implementing this method, know that this is your last chance to do anything before **`self`** disappears. Here is where we make our **`removeObserver:forKeyPath:`** call because once this method completes, **`BLCImagesTableViewController`** will no longer exist and therefore will not need the notifications.

> *Forgetting to remove observers after deallocation may result in a crash if the observed object attempts to communicate with an observer that no longer exists!*

Handling Key-Value Notifications

All KVO notifications are sent to precisely one method. This is where it could get messy if your class needed to monitor multiple keys. However, we just have the one that we care about - **`mediaItems`** - so let's begin adding code to handle it:

BLCImagesTableViewController.m

```
+ - (void) observeValueForKeyPath:(NSString *)keyPath ofObject:(id)object change:(NSDictionary *)change context:(void *)context {
+     if (object == [BLCDataSource sharedInstance] && [keyPath isEqualToString:@"mediaItems"]) {
       // Nothing… YET
+     }
+ }
```

So far you see the method signature; it's implemented in **`NSObject`** just like **`addObserver:forKeyPath:`** and **`removeObserver:forKeyPath:`**. B overriding it, we can handle KVO updates. The first thing we do is create an **`if`** statement which checks two things:

- Is this update coming from the **`BLCDataSource`** object we registered with?
- Is **`mediaItems`** the updated key?

If, for whatever reason, these two conditions are not met then we may have received this update in error. Inside the **`if`** block, we're certain that we've received an update we care about. Let's venture forward and add some code to handle the update:

BLCImagesTableViewController.m

```
- (void) observeValueForKeyPath:(NSString *)keyPath ofObject:(id)object change:(NSDictionary *)change context:(void *)context {
    if (object == [BLCDataSource sharedInstance] && [keyPath isEqualToString:@"mediaItems"]) {
+         // We know mediaItems changed.  Let's see what kind of change it is.
+         int kindOfChange = [change[NSKeyValueChangeKindKey] intValue];

+         if (kindOfChange == NSKeyValueChangeSetting) {
+             // Someone set a brand new images array
+             [self.tableView reloadData];
        }
    }
}
```

All of the relevant data regarding the update is found within the **`change`** dictionary. The first piece of information we'd like to recover is the **kind of change** which occurred. The kinds of changes which can occur are as follows:

Change	Purpose	Constant
`NSKeyValueChangeSetting`	The entire object has been replaced e.g. **`_mediaItems = [NSMutableArray array];`**	**1**
`NSKeyValueChangeInsertion`	An object has been added to the collection	**2**
`NSKeyValueChangeRemoval`	An object has been removed from the collection	**3**
`NSKeyValueChangeReplacement`	An object has been replaced within the collection	**4**

When an **`NSKeyValueChangeSetting`** occurs, we want to invoke **`reloadData`** on our table view. **`reloadData`** tells the table to scrap its current set of cells and cached information. As a result, it effectively asks for all the information again and rebuilds all of the necessary cells.

However, when an item is inserted, removed or replaced, we have a more efficient and attractive method for updating our table:

BLCImagesTableViewController.m

```
    if (object == [BLCDataSource sharedInstance] && [keyPath isEqualToString:@"mediaItems"]) {
        // We know mediaItems changed.  Let's see what kind of change it is.
        int kindOfChange = [change[NSKeyValueChangeKindKey] intValue];

        if (kindOfChange == NSKeyValueChangeSetting) {
            // Someone set a brand new images array
            [self.tableView reloadData];
-         }
+         } else if (kindOfChange == NSKeyValueChangeInsertion ||
+                    kindOfChange == NSKeyValueChangeRemoval ||
+                    kindOfChange == NSKeyValueChangeReplacement) {
+             // We have an incremental change: inserted, deleted, or replaced images
+
+             // Get a list of the index (or indices) that changed
+             NSIndexSet *indexSetOfChanges = change[NSKeyValueChangeIndexesKey];
+
+             // Convert this NSIndexSet to an NSArray of NSIndexPaths (which is what the table view animation methods require)
+             NSMutableArray *indexPathsThatChanged = [NSMutableArray array];
+             [indexSetOfChanges enumerateIndexesUsingBlock:^(NSUInteger idx, BOOL *stop) {
+                 NSIndexPath *newIndexPath = [NSIndexPath indexPathForRow:idx inSection:0];
+                 [indexPathsThatChanged addObject:newIndexPath];
+             }];
+
+             // Call `beginUpdates` to tell the table view we're about to make changes
+             [self.tableView beginUpdates];
+
+             // Tell the table view what the changes are
+             if (kindOfChange == NSKeyValueChangeInsertion) {
+                 [self.tableView insertRowsAtIndexPaths:indexPathsThatChanged withRowAnimation:UITableViewRowAnimationAutomatic];
+             } else if (kindOfChange == NSKeyValueChangeRemoval) {
+                 [self.tableView deleteRowsAtIndexPaths:indexPathsThatChanged withRowAnimation:UITableViewRowAnimationAutomatic];
+             } else if (kindOfChange == NSKeyValueChangeReplacement) {
+                 [self.tableView reloadRowsAtIndexPaths:indexPathsThatChanged withRowAnimation:UITableViewRowAnimationAutomatic];
+             }
+
+             // Tell the table view that we're done telling it about changes, and to complete the animation
+             [self.tableView endUpdates];
+         }
+     }
}
```

A lot is going on in this **`else if`** block, let's step through it. Our **`else if`** checks to make sure that the change which occurred is one of the remaining options: insert, remove or replace. From there, we recover an **`NSIndexSet`** which represents every index in the array which has been modified. For example, if images **`2`** and **`3`** were removed from **`mediaItems`**, those two values would be found in this set.

```
NSIndexSet *indexSetOfChanges = change[NSKeyValueChangeIndexesKey];

NSMutableArray *indexPathsThatChanged = [NSMutableArray array];
[indexSetOfChanges enumerateIndexesUsingBlock:^(NSUInteger idx, BOOL *stop) {
    NSIndexPath *newIndexPath = [NSIndexPath indexPathForRow:idx inSection:0];
    [indexPathsThatChanged addObject:newIndexPath];
}];
```

When updating the rows in a table view, the table requires an **`NSArray`** of **`NSIndexPath`** objects, the kind which have both a **`section`** and **`row`** property. Thankfully, all of our rows are in a single section and are ordered by their location in the **`mediaItems`** array. Therefore, we can simpl loop through **`indexSetOfChanges`** and add to a brand new array, **`indexPathsThatChanged`**.

```
[self.tableView beginUpdates];
if (kindOfChange == NSKeyValueChangeInsertion) {
    [self.tableView insertRowsAtIndexPaths:indexPathsThatChanged withRowAnimation:UITableViewRowAnimationAutomatic];
} else if (kindOfChange == NSKeyValueChangeRemoval) {
    [self.tableView deleteRowsAtIndexPaths:indexPathsThatChanged withRowAnimation:UITableViewRowAnimationAutomatic];
} else if (kindOfChange == NSKeyValueChangeReplacement) {
    [self.tableView reloadRowsAtIndexPaths:indexPathsThatChanged withRowAnimation:UITableViewRowAnimationAutomatic];
}
[self.tableView endUpdates];
```

This lets the table accumulate the modifications we pass to it in preparation for when we call its complimentary method, **`endUpdates`**. Once **`endUpdates`** is invoked, the table takes all of the changes made to its underlying structure since **`beginUpdates`** was called and animates itself to represent the new structure.

Allowing Deletion in our Data Source

So far we've handled deletes, insertions and updates which we find out about thanks to KVO. But how will any class be able to modify the array if it's trapped inside of **`BLCDataSource`**? Let's add a method to **`BLCDataSource`** which lets other classes delete a media item:

BLCDataSource.h

```
+ @class BLCMedia;

@interface BLCDataSource : NSObject

+ (instancetype) sharedInstance;

@property (nonatomic, strong, readonly) NSArray *mediaItems;

+ - (void) deleteMediaItem:(BLCMedia *)item;

@end
```

BLCDataSource.m

```
+ - (void) deleteMediaItem:(BLCMedia *)item {
+     NSMutableArray *mutableArrayWithKVO = [self mutableArrayValueForKey:@"mediaItems"];
+     [mutableArrayWithKVO removeObject:item];
+ }
```

The implementation for this method may seem odd given you know that **`BLCDataSource`** has a reference to **`_mediaItems`**. Why do we use **`mutableArrayValueForKey:`** instead of modifying the **`_mediaItems`** array directly? This is done so KVO updates are sent to the observers. If w remove the item from our underlying data source without going through KVC methods, no one (including **`BLCImagesTableViewController`**) will receive an update.

Enabling swipe-to-delete

Lastly, we want to enable that suave *swipe-to-delete* gesture which you find in most apps today. This is remarkably simple to achieve because the table view supports it natively. Knowing this you can imagine why very many applications tend to implement this common gesture:

BLCImagesTableViewController.m

```
+ - (void)tableView:(UITableView *)tableView commitEditingStyle:(UITableViewCellEditingStyle)editingStyle forRowAtIndexPath:(NSIndexP
+ {
+     if (editingStyle == UITableViewCellEditingStyleDelete) {
+         // Delete the row from the data source
+         BLCMedia *item = [BLCDataSource sharedInstance].mediaItems[indexPath.row];
+         [[BLCDataSource sharedInstance] deleteMediaItem:item];
+     }
+ }
```

And we're done! Run your app, and swipe-to-delete should work. Try deleting images from the top, middle, and bottom of the table to see ho the animation differs.

Not only does swipe-to-delete work, but our data source is prepared for adding and replacing images as well.

- Read Apple's **Introduction to Key-Value Observing Programming Guide**.
- Ask your mentor any questions you have.

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

Terminal

```
$ git add .
$ git commit -m 'KVO and swipe to delete added'
$ git checkout master
$ git merge key-value-observation
$ git push
```

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

assignment completed

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

Infinite Scroll and Pull to Refresh

> *"To infinity... and beyond!"*
>
> *- Buzz Lightyear, fictional character*

Pull to Refresh and Infinite Scroll

We're almost ready to connect to Instagram and download images. But first, we need to make our app support pull-to-refresh and infinite scrolling.

Pull-to-Refresh: A gesture where pulling down on the top of a table view initiates the refreshing of its contents.

Infinite Scroll: When the user reaches the bottom of a table view, attempt to load older content so the user can continue scrolling down towards the beginning of time.

In order for either of these two features to function properly, we're going to need some way for our data source to notify us when new items are available. So far, we've learned two general ways which objects may communicate with one another:

- Delegation protocols
- Key-Value Observing

We'll now learn a third, *completion handlers.*

Begin by creating a new branch:

Terminal

```
$ git co -b pull-to-refresh-infinite-scroll
```

Defining a Completion Handler

Completion handlers are custom code blocks. We will use these blocks as method parameters. For example, **`getLatestImagesWithCompletionBlock:`** would accept a block and **`BLCDataSource`** will execute that block once the images have been recovered.

Completion handlers are meant for asynchronous operations, e.g. downloading something from the internet, waiting on a user response, etc Any method which takes an indeterminate amount of time to complete may benefit from employing a completion handler.

Let's add a new completion handler definition to our data source's header:

```
@class BLCMedia;

 + typedef void (^BLCNewItemCompletionBlock)(NSError *error);

@interface BLCDataSource : NSObject
```

- **typedef** You've encountered **typedef** or *type definition* before. It allows us to make symbolic types, e.g. all instances of *this* actually refer to *that*. For example, if we didn't like the type name **long long** for its confusing repetitiveness, we could **typedef** it to something else: **typedef long lnog uberLong**. Henceforth, the following two statements would be equivalent:

```
long long theLongLongIHave = 5;
```

```
uberLong theLongIDeserve = 5;
```

Both variables, **theLongLongIHave** and **theLongIDeserve** are of primitive type **long long** and store the value **5** however, it is my preference t name them **uberLong**s.

We're using **typedef** in order to define a block which we can reuse as a parameter in multiple methods. If your block is used exclusively in on method, it's best to define it in the signature. However, that won't be the case for **BLCDataSource** and the block we've just defined.

- **void (^BLCNewItemCompletionBlock)** The next portion of our definition does two things: it specifies the return type required of the block, *void,* and gives it a name, **BLCNewItemCompletionBlock**. Just follow these syntactical rules and you'll be able to define your own blocks in the future.

- **(NSError *error);** The last piece of the type definition is the block's only parameter. In **NSArray**'s **enumerateObjectsUsingBlock:** method, the block parameters are **(id obj, NSUInteger idx, BOOL *stop)**. These parameters can be of any type or length, just like a method. We've chosen an **NSError** object as our parameter. This is a typical pattern followed by completion handlers.

If the **error** object comes back as anything other than **nil**, something has gone wrong during the execution of that method. An **NSError** is capable of providing an error code, **code**, a human-readable string describing the error, **localizedDescription** and an **NSDictionary** of additional error-related information, **userInfo**. Errors like network failure, bad access, request time out, etc. may be reported back to the use through **NSError** objects.

Fetching New Images

We're going to need a method which the table view will call if and when the user executes a pull-to-refresh gesture. Since the method is mear to be asynchronous, we're going to require that the caller passes in a version of our completion handler, **BLCNewItemCompletionBlock**:

BLCDataSource.h

```
@interface BLCDataSource : NSObject

+ (instancetype) sharedInstance;

@property (nonatomic, strong, readonly) NSArray *mediaItems;

- (void) deleteMediaItem:(BLCMedia *)item;

 + - (void) requestNewItemsWithCompletionHandler:(BLCNewItemCompletionBlock)completionHandler;

@end
```

In the implementation file, we're going to add a **BOOL** property to track whether a refresh is already in progress. We don't want our method to fetch multiple times while it's waiting for a result from the Instagram servers. Our current implementation is entirely local and does not acces the internet but it will in the coming checkpoints:

BLCDataSource.m

```
    NSMutableArray *_mediaItems;
}

@property (nonatomic, strong) NSArray *mediaItems;

+ @property (nonatomic, assign) BOOL isRefreshing;

@end
```

BLCDataSource.m

```
+ - (void) requestNewItemsWithCompletionHandler:(BLCNewItemCompletionBlock)completionHandler {
+     if (self.isRefreshing == NO) {
+         self.isRefreshing = YES;
+         BLCMedia *media = [[BLCMedia alloc] init];
+         media.user = [self randomUser];
+         media.image = [UIImage imageNamed:@"10.jpg"];
+         media.caption = [self randomSentenceWithMaximumNumberOfWords:7];

+         NSMutableArray *mutableArrayWithKVO = [self mutableArrayValueForKey:@"mediaItems"];
+         [mutableArrayWithKVO insertObject:media atIndex:0];

+         self.isRefreshing = NO;

+         if (completionHandler) {
+             completionHandler(nil);
+         }
+     }
+ }
```

Let's examine this code briefly. First we check the **BOOL** property we established moments ago. If a request for recovering new items is already in progress, we exit immediately. Otherwise, we set **isRefreshing** to **YES** and continue onwards.

The next section of code should be familiar. Here we simply create a new random media object and append it to the front of our KVC array. We place the media item at index **0** because that is the index which appears at the top-most table cell.

> *In later checkpoints this portion will be replaced with accessing the Instagram API.*

Once that's completed, we reset the **isRefreshing** boolean to **NO** since we are no longer in the process of refreshing. Finally, we check if a completion handler was passed before calling it with **nil**. We do not provide an **NSError** object simply because creating a fake, local piece of data like **media** will rarely result in an issue. The **NSError** will be employed once we begin communicating with Instagram.

Adding Pull-to-Refresh

All that's left is to support the pull-to-refresh gesture. Thankfully it's been built into the **UITableViewController** since iOS 6.0. In order to support it we really won't have to add much:

BLCImagesTableViewController.m

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    [[BLCDataSource sharedInstance] addObserver:self forKeyPath:@"mediaItems" options:0 context:nil];

+     self.refreshControl = [[UIRefreshControl alloc] init];
+     [self.refreshControl addTarget:self action:@selector(refreshControlDidFire:) forControlEvents:UIControlEventValueChanged];

    [self.tableView registerClass:[BLCMediaTableViewCell class] forCellReuseIdentifier:@"mediaCell"];
}
```

UITableViewController and all we needed to do was initialize it and set a target for **UIControlEventValueChanged**. Let's fill in the method w chose, **refreshControlDidFire:**:

BLCImagesTableViewController.m

```
+ - (void) refreshControlDidFire:(UIRefreshControl *) sender {
+     [[BLCDataSource sharedInstance] requestNewItemsWithCompletionHandler:^(NSError *error) {
+         [sender endRefreshing];
+     }];
+ }
```

> *Try typing out the code snippet above by hand in order to experience Xcode's block auto-completion:*
>
> 
>

As you can see, we invoke our brand new data source method, **requestNewItemsWithCompletionHandler:** with a completion handler in tow. When our handler is invoked by the line we wrote earlier:

BLCDataSource.m

```
if (completionHandler) {
    completionHandler(nil);
}
```

Our code block is then executed. The only thing our code block does (**[sender endRefreshing];**) is tell the **UIRefreshControl** to stop spinning and hide itself.

Now run your app. When you pull-to-refresh, a new image should appear at the top.

Adding Infinite Scroll

Supporting infinite scroll will look and behave similarly to the previous work we've done for pull-to-refresh. The new method we'll add to **BLCDataSource** should be self-explanatory, it even uses the same completion handler we defined earlier:

BLCDataSource.h

```
@interface BLCDataSource : NSObject

+ (instancetype) sharedInstance;

@property (nonatomic, strong, readonly) NSArray *mediaItems;

- (void) deleteMediaItem:(BLCMedia *)item;

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

@end
```

BLCDataSource.m

```
    NSMutableArray *_mediaItems;
}

@property (nonatomic, strong) NSArray *mediaItems;

@property (nonatomic, assign) BOOL isRefreshing;
+ @property (nonatomic, assign) BOOL isLoadingOlderItems;

@end
```

BLCDataSource.m

```
+ - (void) requestOldItemsWithCompletionHandler:(BLCNewItemCompletionBlock)completionHandler {
+     if (self.isLoadingOlderItems == NO) {
+         self.isLoadingOlderItems = YES;
+         BLCMedia *media = [[BLCMedia alloc] init];
+         media.user = [self randomUser];
+         media.image = [UIImage imageNamed:@"1.jpg"];
+         media.caption = [self randomSentenceWithMaximumNumberOfWords:7];

+         NSMutableArray *mutableArrayWithKVO = [self mutableArrayValueForKey:@"mediaItems"];
+         [mutableArrayWithKVO addObject:media];

+         self.isLoadingOlderItems = NO;

+         if (completionHandler) {
+             completionHandler(nil);
+         }
+     }
+ }
```

The changes made to **`BLCDataSource`** look and behave identically to those we made earlier in order to support pull-to-refresh. However, the changes we need to make to **`BLCImagesTableViewController`** will require something different than before:

BLCImagesTableViewController.m

```
+ - (void) infiniteScrollIfNecessary {
+     NSIndexPath *bottomIndexPath = [[self.tableView indexPathsForVisibleRows] lastObject];

+     if (bottomIndexPath && bottomIndexPath.row == [BLCDataSource sharedInstance].mediaItems.count - 1) {
+         // The very last cell is on screen
+         [[BLCDataSource sharedInstance] requestOldItemsWithCompletionHandler:nil];
+     }
+ }

+ #pragma mark - UIScrollViewDelegate

+ - (void)scrollViewDidScroll:(UIScrollView *)scrollView {
+     [self infiniteScrollIfNecessary];
+ }
```

Our first method, **`infiniteScrollIfNecessary`** checks whether or not the user has scrolled to the last photo. This is accomplished by recovering an array of **`NSIndexPath`** objects which represent the cells visible on screen. We then call **`lastObject`** on the **`NSArray`** returned by **`indexPathsForVisibleRows`** in order to recover the index path of the cell shown at the very bottom of the table.

Finally, if that cell represents the last image in the **`_mediaItems`** array, we call **`requestOldItemsWithCompletionHandler:`** in order to recover more.

The next method needs a little more explanation. **`UITableView`** is a subclass of **`UIScrollView`**. A scroll view, plainly stated, is a UI element which scrolls. Its content size can be substantially larger than its frame, and a dragging gesture is capable of navigating its content. A scroll view can be scrolled horizontally and/or vertically. (A table view is locked into vertical-only scrolling.)

The scroll view has a delegate protocol with many methods in it, one of which is **`scrollViewDidScroll:`**. This delegate method is invoked if and when the scroll view is scrolled in any direction. As the user scrolls the table view, this method will be called repeatedly. It's a good place to check whether or not the last image in our array has made it onto the screen.

- Read Apple's **Conceptual Overview of Blocks**.
- See if you can use a different **UIScrollView delegate** method to reduce the number of times we call `infiniteScrollIfNecessary`.
- Ask your mentor any questions you have.

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 pull-to-refresh support and infinite scrolling'
$ git checkout master
$ git merge pull-to-refresh-infinite-scroll
$ git push
```

assignment completed

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

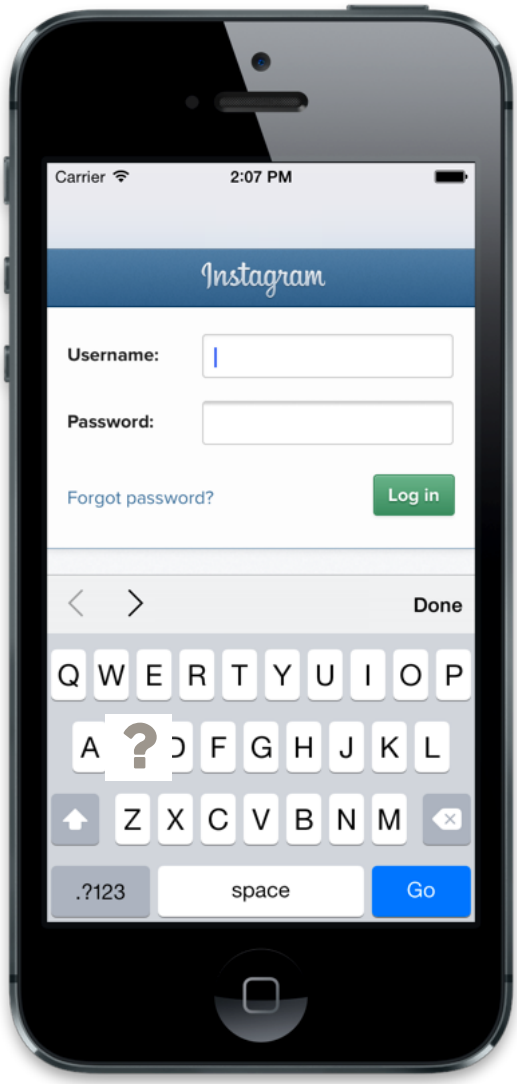
hello@bloc.io

Considering enrolling? (404) 480-2562

Partnership / Corporate Inquiries? (650) 741-5682

Roadmap · Previous Checkpoint
Jump to Submission · Next Checkpoir

Connecting to Instagram: Logging In



It's time to connect our app to Instagram! We'll start by logging in.

Before we do that, we'll need to register for the Instagram API.

> *API is an acronym for Application Programming Interface. An API is any specification for how two different pieces of software communicate.*

Register for the Instagram API

- You can make your "OAuth redirect_uri" anything you want. We used "**http://bloc.io**".
- "Disable implicit OAuth" and "Enforce signed header" should be unchecked.

You should now have a "Client ID" and a "Client Secret". Keep those in a safe place.

Now let's work on some code. Create a new branch:

Terminal

```
$ git checkout -b instagram-login
```

Store your "Client ID" in **`BLCDataSource`**.

BLCDataSource.h

```
+ + (NSString *) instagramClientID;
```

BLCDataSource.m

```
+ + (NSString *) instagramClientID {
+     return @"<your client ID>";
+ }
```

Allowing Users to Login to Instagram

Before we can view a user's content, we'll need to login. To accomplish this, we'll make a login view controller.

Once users login, Instagram appends a string called an “access token” to the “redirect URI“ you specified earlier. (This will make more sense when we get to the code below.)

An **access token** is like a password, just for our app: it's a string that encodes security credentials for a user's Instagram account. It's specific to our application and the privileges that the user grants us.

Make a new **`UIViewController`** subclass called **`BLCLoginViewController`**.

The **`.h`** file will be short and sweet:

BLCLoginViewController.h

```
+ @interface BLCLoginViewController : UIViewController
+
+ extern NSString *const BLCLoginViewControllerDidGetAccessTokenNotification;
+
+ @end
```

We're declaring a constant string called **`BLCLoginViewControllerDidGetAccessTokenNotification`**. As you'll see later, anyone who wants to be notified when an access token is obtained will use this string.

Now let's create our **`.m`** file.

We'll use a **`UIWebView`** for the user to login. Let's make a property for it:

BLCLoginViewController.m

```
+
+ @interface BLCLoginViewController () <UIWebViewDelegate>
+
+ @property (nonatomic, weak) UIWebView *webView;
+
+ @end
```

We also declare that we'll conform to the **`UIWebViewDelegate`** protocol.

In our implementation, we'll store a value in that **`extern`** string we declared in the **`.h`** file:

BLCLoginViewController.m

```
+ @implementation BLCLoginViewController
+
+ NSString *const BLCLoginViewControllerDidGetAccessTokenNotification = @"BLCLoginViewControllerDidGetAccessTokenNotification";
```

We're making the string equal to its name. We don't have to do this; it's just a convention. (Setting it to **`@"Birthday cake is delicious"`** would work just as well.) You'll see how we use this string in a bit.

Add a method to store the redirect URI you chose:

BLCLoginViewController.m

```
+ - (NSString *)redirectURI {
+     return @"<the redirect URI you specified when signing up with instagram>";
+ }
```

Create the web view and set its delegate:

BLCLoginViewController.m

```
+ - (void)loadView {
+     UIWebView *webView = [[UIWebView alloc] init];
+     webView.delegate = self;
+
+     self.webView = webView;
+     self.view = webView;
+ }
```

Now we'll need to send the user to a special Instagram page to login. Once the user logs in, we'll get their access token.

In order to get the correct login page, you'll need to provide the Instagram Client ID provided when you registered your client, which we store in **`BLCDataSource`**.

BLCLoginViewController.m

```
+ - (void)viewDidLoad
+ {
+     [super viewDidLoad];
+     // Do any additional setup after loading the view.
+
+     NSString *urlString = [NSString stringWithFormat:@"https://instagram.com/oauth/authorize/?client_id=%@&redirect_uri=%@&response
+     NSURL *url = [NSURL URLWithString:urlString];
+
+     if (url) {
+         NSMutableURLRequest *request = [NSMutableURLRequest requestWithURL:url];
+         [self.webView loadRequest:request];
+     }
+ }
```

That code will generate and load the appropriate login site in the web view.

1. We need to set the web view's delegate to **nil** (this is a **quirk of** `UIWebView`; most objects don't require this).
2. We want to clear cookies set by the Instagram website.

We can accomplish both of these in **dealloc**:

BLCLoginViewController.m

```
+ - (void) dealloc {
+     // Removing this line causes a weird flickering effect when you relaunch the app after logging in, as the web view is briefly d
+     [self clearInstagramCookies];
+
+     // see https://developer.apple.com/library/ios/documentation/uikit/reference/UIWebViewDelegate_Protocol/Reference/Reference.htm
+     self.webView.delegate = nil;
+ }
+
+ /**
+  Clears Instagram cookies. This prevents caching the credentials in the cookie jar.
+  */
+ - (void) clearInstagramCookies {
+     for(NSHTTPCookie *cookie in [[NSHTTPCookieStorage sharedHTTPCookieStorage] cookies]) {
+         NSRange domainRange = [cookie.domain rangeOfString:@"instagram.com"];
+         if(domainRange.location != NSNotFound) {
+             [[NSHTTPCookieStorage sharedHTTPCookieStorage] deleteCookie:cookie];
+         }
+     }
+ }
```

NSHTTPCookieStorage (sometimes informally called "the cookie jar") stores and manages web site cookies, which are represented as **NSHTTPCookie** objects.

Once you get this code working, try commenting out the call to **[self clearInstagramCookies]** and see what happens differently between app launches.

Now it's time to get the access token, using the web view delegate.

We'll check in one of the web view's delegate methods for a redirection beginning with our redirect URI. If it's there, extract the access token and stop the web view from loading:

BLCLoginViewController.m

```
+ - (BOOL) webView:(UIWebView *)webView shouldStartLoadWithRequest:(NSURLRequest *)request navigationType:(UIWebViewNavigationType)na
+     NSString *urlString = request.URL.absoluteString;
+     if ([urlString hasPrefix:[self redirectURI]]) {
+         // This contains our auth token
+         NSRange rangeOfAccessTokenParameter = [urlString rangeOfString:@"access_token="];
+         NSUInteger indexOfTokenStarting = rangeOfAccessTokenParameter.location + rangeOfAccessTokenParameter.length;
+         NSString *accessToken = [urlString substringFromIndex:indexOfTokenStarting];
+         [[NSNotificationCenter defaultCenter] postNotificationName:BLCLoginViewControllerDidGetAccessTokenNotification object:acces
+         return NO;
+     }
+
+     return YES;
+ }
 @end
```

This method searches for a URL containing the redirect URI, and then sets the access token to everything after **access_token=**.

For example, if your redirect URL is **http://bloc.io**, Instagram could send you to **http://bloc.io?access_token=MY_TOP_SECRET_TOKEN**. We'd then set **accessToken** to **MY_TOP_SECRET_TOKEN**, and post the notification.

> *We're using a new method of communication here called a notification. A notification is a one-to-many form of communication like key-value observing, but less formal. One object (in our case, the login view controller) sends a communication to anyone who's registered when something noteworthy happens (in our case, when an access token is obtained).*

> - **`NSNotificationCenter`** *has nothing to do with the iOS "notification center" where users see texts and push notifications*
> - **`[NSNotificationCenter defaultCenter]`** *is another example of the singleton pattern mentioned earlier*

Responding to the Notification

Now a user can login, but once we get the notification, nothing will happen. To trigger an action based on the notification, we need to register for it and respond to it.

In **`BLCDataSource`**

We need to hang on to the access token, so let's create a property to store it:

BLCDataSource.h

```
+ @property (nonatomic, strong, readonly) NSString *accessToken;
```

BLCDataSource.m

```
+ #import "BLCLoginViewController.h"

@interface BLCDataSource () {
    NSMutableArray *_mediaItems;
}

+ @property (nonatomic, strong) NSString *accessToken;
```

Let's register and respond to the notification, and at the same time, delete the call to the method that generates random data:

BLCDataSource.m

```
- (instancetype) init {
    self = [super init];

    if (self) {
-         [self addRandomData];
+         [self registerForAccessTokenNotification];
    }

    return self;
}

+ - (void) registerForAccessTokenNotification {
+     [[NSNotificationCenter defaultCenter] addObserverForName:BLCLoginViewControllerDidGetAccessTokenNotification object:nil queue:n
+         self.accessToken = note.object;
+     }];
+ }
```

Immediately after the login controller posts the **`BLCLoginViewControllerDidGetAccessTokenNotification`** notification, the block provided will run. The object that's passed in the notification is an **`NSString`** containing the access token, so all we do is store it in **`self.accessToken`** when it arrives.

> *Normally, you would also unregister (`removeObserver:…`) for notifications in `dealloc`. However, since `BLCDataSource` is a singleton, it will never get deallocated.*

We'll do a bit more in a moment, but while we're here, let's delete all of our random data methods since we're no longer using them:

BLCDataSource.m

```
-     NSMutableArray *randomMediaItems = [NSMutableArray array];
-
-     for (int i = 1; i <= 10; i++) {
-         NSString *imageName = [NSString stringWithFormat:@"%d.jpg", i];
-         UIImage *image = [UIImage imageNamed:imageName];
-         if (image) {
-             BLCMedia *media = [[BLCMedia alloc] init];
-             media.user = [self randomUser];
-             media.image = image;
-             media.caption = [self randomSentenceWithMaximumNumberOfWords:7];
-
-             NSUInteger commentCount = arc4random_uniform(10);
-             NSMutableArray *randomComments = [NSMutableArray array];
-
-             for (int i  = 0; i <= commentCount; i++) {
-                 BLCComment *randomComment = [self randomComment];
-                 [randomComments addObject:randomComment];
-             }
-
-             media.comments = randomComments;
-
-             [randomMediaItems addObject:media];
-         }
-     }
-
-     self.mediaItems = randomMediaItems;
- }
-
- - (BLCUser *) randomUser {
-     BLCUser *user = [[BLCUser alloc] init];
-
-     user.userName = [self randomStringOfLength:arc4random_uniform(10)];
-
-     NSString *firstName = [self randomStringOfLength:arc4random_uniform(7)];
-     NSString *lastName = [self randomStringOfLength:arc4random_uniform(12)];
-     user.fullName = [NSString stringWithFormat:@"%@ %@", firstName, lastName];
-
-     return user;
- }
-
- - (BLCComment *) randomComment {
-     BLCComment *comment = [[BLCComment alloc] init];
-
-     comment.from = [self randomUser];
-     comment.text = [self randomSentenceWithMaximumNumberOfWords:20];
-
-     return comment;
- }
-
- - (NSString *) randomSentenceWithMaximumNumberOfWords:(NSUInteger) numberOfWords {
-     NSUInteger wordCount = arc4random_uniform(20);
-
-     NSMutableString *randomSentence = [[NSMutableString alloc] init];
-
-     for (int i  = 0; i <= wordCount; i++) {
-         NSString *randomWord = [self randomStringOfLength:arc4random_uniform(12)];
-         if (randomWord.length > 0) {
-             [randomSentence appendFormat:@"%@ ", randomWord];
-         }
-     }
-
-     return randomSentence;
- }
-
- - (NSString *) randomStringOfLength:(NSUInteger) len {
-     NSString *alphabet = @"abcdefghijklmnopqrstuvwxyz";
-
-     NSMutableString *s = [NSMutableString string];
-     for (NSUInteger i = 0U; i < len; i++) {
-         u_int32_t r = arc4random_uniform((u_int32_t)[alphabet length]);
-         unichar c = [alphabet characterAtIndex:r];
```

```
-     return [NSString stringWithString:s];
- }

- (void) requestNewItemsWithCompletionHandler:(BLCNewItemCompletionBlock)completionHandler {
    if (self.isRefreshing == NO) {
        self.isRefreshing = YES;
-         BLCMedia *media = [[BLCMedia alloc] init];
-         media.user = [self randomUser];
-         media.image = [UIImage imageNamed:@"10.jpg"];
-         media.caption = [self randomSentenceWithMaximumNumberOfWords:7];
-
-         NSMutableArray *mutableArrayWithKVO = [self mutableArrayValueForKey:@"mediaItems"];
-         [mutableArrayWithKVO insertObject:media atIndex:0];
+
+         // Need to add images here

        self.isRefreshing = NO;

        if (completionHandler) {
            completionHandler(nil);
        }
    }
}

- (void) requestOldItemsWithCompletionHandler:(BLCNewItemCompletionBlock)completionHandler {
    if (self.isLoadingOlderItems == NO) {
        self.isLoadingOlderItems = YES;
-         BLCMedia *media = [[BLCMedia alloc] init];
-         media.user = [self randomUser];
-         media.image = [UIImage imageNamed:@"1.jpg"];
-         media.caption = [self randomSentenceWithMaximumNumberOfWords:7];
-
-         NSMutableArray *mutableArrayWithKVO = [self mutableArrayValueForKey:@"mediaItems"];
-         [mutableArrayWithKVO addObject:media];
+
+         // Need to add images here

        self.isLoadingOlderItems = NO;

        if (completionHandler) {
            completionHandler(nil);
        }
    }
}
```

Now, we'll use a similar notification-based approach to update the correct view controller in the App Delegate.

In the App Delegate

First, import the two classes we just updated:

BLCAppDelegate.m

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

Our app should use this logic:

- At launch, show the login controller
- Register for the **`BLCLoginViewControllerDidGetAccessTokenNotification`** notification
- When this notification posts, switch the root view controller from the login controller to the table controller

BLCAppDelegate.m

```
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
-     self.window.rootViewController = [[UINavigationController alloc] initWithRootViewController:[[BLCImagesTableViewController allo
+
+     [BLCDataSource sharedInstance]; // create the data source (so it can receive the access token notification)
+
+     UINavigationController *navVC = [[UINavigationController alloc] init];
+     BLCLoginViewController *loginVC = [[BLCLoginViewController alloc] init];
+     [navVC setViewControllers:@[loginVC] animated:YES];
+
+     [[NSNotificationCenter defaultCenter] addObserverForName:BLCLoginViewControllerDidGetAccessTokenNotification object:nil queue:n
+         BLCImagesTableViewController *imagesVC = [[BLCImagesTableViewController alloc] init];
+         [navVC setViewControllers:@[imagesVC] animated:YES];
+     }];
+
+     self.window.rootViewController = navVC;
```

This will start the app with the login view controller, and switch to the images table controller once an access token is obtained.

Getting media data from Instagram

It would be a shame to end the checkpoint here, without actually *using* the access token. So let's do the very beginning of what should theoretically be in the next checkpoint - get a list (or "feed") of the user's images.

According to the **Instagram API reference** we can get the user's image feed by accessing this URL:

```
https://api.instagram.com/v1/users/self/feed?access_token=<access_token>
```

In addition to **access_token**, we can optionally specify these other parameters:

> ***COUNT***: *Count of media to return.*
> ***MIN_ID***: *Return media later than this min_id.*
> ***MAX_ID***: *Return media earlier than this max_id.*

Let's write a method to create this request, and turn the response from the Instagram API into a dictionary:

BLCDataSource.m

```
+     if (self.accessToken) {
+         // 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]
+
+                 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];
+                     });
+                 }
+             }
+         });
+     }
+ }
+
+ - (void) parseDataFromFeedDictionary:(NSDictionary *) feedDictionary fromRequestWithParameters:(NSDictionary *)parameters {
+     NSLog(@"%@", feedDictionary);
+ }
```

There's a lot of new stuff here. Here is a brief explanation of each new class or method.

`dispatch_async`

When you have long-running work, like network connections or complex calculations, you should do that work in the background. This allows your user interface - which always runs on the main queue - to remain responsive.

A common pattern - the one you see here - is to **`dispatch_async`** on to a background queue, and then when the long-running work is completed, **`dispatch_async`** back on to the main queue.

`NSURLConnection`

So far, when we've had an **`NSURLRequest`**, we've given it to a **`UIWebView`** for loading, rendering and displaying. When you don't want to directly display the data, you can use **`NSURLConnection`** to handle connecting to a server and downloading the data.

`NSData`

`NSData` is an object that represents any type of data. If the data is an image, you can convert it into a **`UIImage`**. If it's a string, you can convert i into an **`NSString`**.

Passing addresses into methods

In this code, we pass *addresses* into methods. This happens in three places: **`&response`**, **`&webError`**, and **`&jsonError`**.

For example, in this line:

```
NSData *responseData = [NSURLConnection sendSynchronousRequest:request returningResponse:&response error:&webError];
```

NSURLConnection returns an **NSData** object. But it also wants to communicate some other information - metadata about the response (**NSURLResponse**) and possibly an error, if something went wrong (**NSError**).

Since Objective-C can only return 1 method, we pass in addresses of other variables as arguments, and the method sets them.

> *This is commonly called "vending" - we would say that* **NSURLConnection***'s method returns an* **NSData** *and vends an* **NSURLRequest** *and an* **NSError**.

JSON and **NSJSONSerialization**

On the **Instagram API page**, press the "response" button to see a sample of what the response data looks like.

The format of this data is called JSON, which is a way to organize strings, numbers, arrays, and dictionaries using standard symbols.

NSJSONSerialization is a class that converts this data into the more familiar **NSDictionary** and **NSArray** objects.

> *Serialization is the process of converting data from one format to another.*

For more info on JSON, see Bloc's **Intro to Networking for Mobile Developers**.

- Now that we have our actual data, let's take a look at it. Add a call to this method once our access token arrives:

BLCDataSource.m

```
- (void) registerForAccessTokenNotification {
    [[NSNotificationCenter defaultCenter] addObserverForName:BLCLoginViewControllerDidGetAccessTokenNotification object:nil queue:nil
        self.accessToken = note.object;

+         // Got a token, populate the initial data
+         [self populateDataWithParameters:nil];
    }];
}
```

Run the app. After logging in, the output will be logged to the console.

In the next checkpoint, we'll use this data to actually display the images on the screen.

Summary

We covered a number of new topics in this checkpoint:

- Using **NSNotificationCenter** to send one-to-many communications between objects
- Using **dispatch_async** to execute long-running code in the background
- Using **NSURLConnection** to download data
- Using **NSJSONSerialization** to convert the data into information stored in **NSDictionary** and **NSArray** objects
- Working with methods that return more than one value by "vending" objects

While this seems like a ton of stuff, it's important to remember that **these concepts are commonly used together, often in the exact sam way**. As we do other things with the Instagram API, you'll start to understand how they work together better.

- Make the navigation bar's title say "Login" when the user is logging in
- In the login window, if the user clicks "Forgot Password", and then "About Us" in the footer, there's no way to get back to the login page. Fix this by adding either a "back" or "home" button to the navigation bar.

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

Terminal

```
$ git add .
$ git commit -m 'Add Instagram Login'
$ git checkout master
$ git merge instagram-login
$ git push
```

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

assignment completed

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

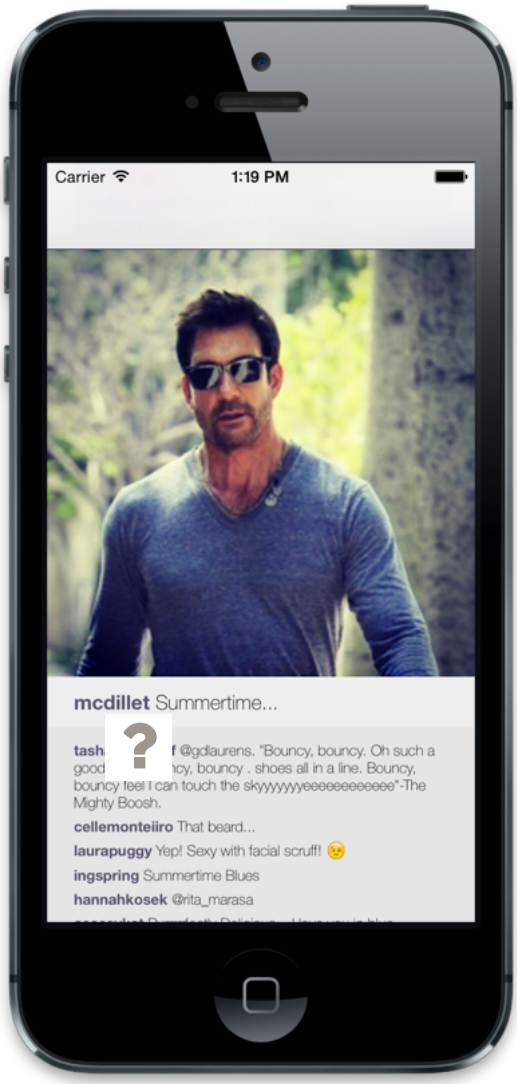
hello@bloc.io

Considering enrolling? (404) 480-2562

Partnership / Corporate Inquiries? (650) 741-5682

Roadmap · Previous Checkpoint
Jump to Submission · Next Checkpoir

Connecting to Instagram: Retrieving Images



> *"Life is like an onion. You peel it off one layer at a time, and sometimes you weep."*
>
> *- Carl Sandburg, Pulitzer-prize-winning poet, author of Abraham Lincoln biography, and folk singer*

In this checkpoint, we'll:

- parse the JSON that we obtained from the Instagram API,
- use it to create **`BLCMedia`**, **`BLCUser`**, and **`BLCComment`** objects
- download the images, and
- display them in the table view

Create a new branch:

```
$ git checkout -b instagram-images
```

Onion-Peelin'

Parsing JSON is a lot like peeling an onion: remove one layer at a time until you get what you want.

Consider this excerpt of our JSON:

```
{
  "pagination": {

  },
  "meta": {
    "code": 200
  },
  "data": [
    {
      "attribution": null,
      "tags": [

      ],
      "type": "image",
      "location": null,
      "comments": {
        "count": 84,
        "data": [
          {
            "created_time": "1401358234",
            "text": "OMG!",
            "from": {
              "username": "fafa313",
              "profile_picture": "http:\/\/images.ak.instagram.com\/profiles\/profile_582637650_75sq_1399534296.jpg",
              "id": "582637650",
              "full_name": "\u0641\u0627\u0641\u0627"
            },
            "id": "730968308560302306"
          },
          {
            "created_time": "1401374997",
            "text": "U know u got it",
            "from": {
              "username": "lutherpics",
              "profile_picture": "http:\/\/images.ak.instagram.com\/profiles\/profile_309451969_75sq_1372650775.jpg",
              "id": "309451969",
              "full_name": "Meshelly Luther"
            },
            "id": "731108930889847663"
          },
```

Let's say we wanted to get the **`profile_picture`** from the second comment (lutherpics's profile picture). We can do something like this:

```
NSDictionary *startDictionary = … // the JSON
NSArray *dataArray = startDictionary[@"data"];
NSDictionary *firstPost = dataArray[0];
NSDictionary *commentDictionary = firstPost[@"comments"];
NSArray *commentDataArray = commentDictionary[@"data"];
NSDictionary *secondCommentDictionary = commentDataArray[1];
NSDictionary *fromDictionary = secondCommentDictionary[@"from"];
NSString *profilePicture = fromDictionary[@"profile_picture"];
```

You can also string the subscripts together, like this:

> ***Common bug warning.*** *If you mix up an* **`NSDictionary`** *and* **`NSArray`** *you'll cause a crash, because dictionaries don't have indexes and arrays don't have keys.*
>
> - **`NSDictionary`** *uses keys, so use keyed subscripts (*`[@"like this"]`*)*
> - **`NSArray`** *uses indexes, so use indexed subscripts (*`[0]`*,* `[1]`*,* `[2]`*, etc.)*

Since we have so much JSON, we'll break it up, and make each object responsible for parsing its own layer of the JSON onion.

Adding initializers that parse

Adding lots of JSON-parsing code in one data source class (like **`BLCDataSource`**) is a common practice, but it has two problems:

1. It would make your data source class very large as you start to parse more and more types of data.
2. It would unnecessarily couple your model object (**`BLCUser`**, for example) with your data source class. This makes your code less flexible.

Instead, we can make model objects responsible for parsing the section of JSON that they represent.

Users

For example, to create a **`BLCUser`**, we'll add a new method called **`initWithDictionary:`**:

BLCUser.h

```
+ - (instancetype) initWithDictionary:(NSDictionary *)userDictionary;
```

BLCUser.m

```
+ - (instancetype) initWithDictionary:(NSDictionary *)userDictionary {
+     self = [super init];
+
+     if (self) {
+         self.idNumber = userDictionary[@"id"];
+         self.userName = userDictionary[@"username"];
+         self.fullName = userDictionary[@"full_name"];
+
+         NSString *profileURLString = userDictionary[@"profile_picture"];
+         NSURL *profileURL = [NSURL URLWithString:profileURLString];
+
+         if (profileURL) {
+             self.profilePictureURL = profileURL;
+         }
+     }
+
+     return self;
+ }
```

We'll pass in the part of the Instagram JSON that represents a user:

JSON

```
{
    "username": "lutherpics",
    "profile_picture": "http:\/\/images.ak.instagram.com\/profiles\/profile_309451969_75sq_1372650775.jpg",
    "id": "309451969",
    "full_name": "Meshelly Luther"
}
```

Most of this we're storing in properties for later use, but note that we convert **`profile_picture`** into an **`NSURL`**.

Comments

We'll do something similar with **`BLCComment`**.

BLCComment.h

```
+ - (instancetype) initWithDictionary:(NSDictionary *)commentDictionary;
```

BLCComment.m

```
+ - (instancetype) initWithDictionary:(NSDictionary *)commentDictionary {
+     self = [super init];
+
+     if (self) {
+         self.idNumber = commentDictionary[@"id"];
+         self.text = commentDictionary[@"text"];
+         self.from = [[BLCUser alloc] initWithDictionary:commentDictionary[@"from"]];
+     }
+
+     return self;
+ }
```

Note that a comment looks like this:

JSON

```
{
  "created_time": "1401374997",
  "text": "U know u got it",
  "from": {
    "username": "lutherpics",
    "profile_picture": "http:\/\/images.ak.instagram.com\/profiles\/profile_309451969_75sq_1372650775.jpg",
    "id": "309451969",
    "full_name": "Meshelly Luther"
  },
  "id": "731108930889847663"
}
```

When the **`from`** dictionary is reached, note that **`BLCComment`** will extract it and pass it to **`BLCUser`** for parsing.

> *This is an example of the design principles encapsulation and separation of concerns. If we need to change something about how a user works, we won't need to update* **`BLCComment`**, *because* **`BLCComment`** *is handing off that responsibility to* **`BLCUser`**.

Media

The media parser will look somewhat familiar, but with a bit of new hotness:

BLCMedia.h

```
+ - (instancetype) initWithDictionary:(NSDictionary *)mediaDictionary;
```

BLCMedia.m

```
+     self = [super init];
+
+     if (self) {
+         self.idNumber = mediaDictionary[@"id"];
+         self.user = [[BLCUser alloc] initWithDictionary:mediaDictionary[@"user"]];
+         NSString *standardResolutionImageURLString = mediaDictionary[@"images"][@"standard_resolution"][@"url"];
+         NSURL *standardResolutionImageURL = [NSURL URLWithString:standardResolutionImageURLString];
+
+         if (standardResolutionImageURL) {
+             self.mediaURL = standardResolutionImageURL;
+         }
+
+         NSDictionary *captionDictionary = mediaDictionary[@"caption"];
+
+         // Caption might be null (if there's no caption)
+         if ([captionDictionary isKindOfClass:[NSDictionary class]]) {
+             self.caption = captionDictionary[@"text"];
+         } else {
+             self.caption = @"";
+         }
+
+         NSMutableArray *commentsArray = [NSMutableArray array];
+
+         for (NSDictionary *commentDictionary in mediaDictionary[@"comments"][@"data"]) {
+             BLCComment *comment = [[BLCComment alloc] initWithDictionary:commentDictionary];
+             [commentsArray addObject:comment];
+         }
+
+         self.comments = commentsArray;
+     }
+
+     return self;
+ }
```

A few notes, though:

Images

Instagram's JSON gives us a few different resolutions of images. This would be helpful, for example, if we wanted to build a thumbnail view. For now, we'll just use the **`standard_resolution`** image.

Caption

Instagram's API provides different results depending on whether an image has a caption or not.

If there *is* a caption, the JSON looks like this:

JSON

```
"caption":{
  "created_time":"1343765325",
  "text":"Part of my job to watch swimming. #olympics #USAUSAUSA",
  "from":{
    "username":"kissinkatkelly",
    "profile_picture":"http:\/\/images.instagram.com\/profiles\/profile_4672491_75sq_1341713095.jpg",
    "id":"4672491",
    "full_name":"kissinkatkelly"
  },
  "id":"247843973390332239"
}
```

This is serialized as an **`NSDictionary`**, and we just get the **`text`** value. (Technically, we could create a new object called **`BLCCaption`** and parse the whole thing, but our current user interface doesn't need any of the other data.)

If there *isn't* a caption, the JSON looks like this:

```
"caption": null
```

This is serialized as an `NSNull` object. **`NSNull`** is a rather pointless object. Its only job is to represent **`null`** in collections like **`NSDictionary`** and **`NSArray`**. (Since **`null`** isn't an object, it can't go in Objective-C collections.)

Because **`mediaDictionary[@"caption"]`** can return *either* an **`NSDictionary`** or **`NSNull`**, we must check to ensure we've got the correct type.

Imagine the code simply looked like this:

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

We're assuming that **`captionDictionary`** will be an **`NSDictionary`**. But if it's actually an **`NSNull`**, the second line will cause a crash because **`NSNull`** doesn't have keys like **`NSDictionary`** does.

That's why we check using **`isKindOfClass`**, and avoid using the keyed subscript until we're sure we got the class, **`NSDictionary`**, correct:

```
NSDictionary *captionDictionary = mediaDictionary[@"caption"];
// Caption might be null (if there's no caption)
if ([captionDictionary isKindOfClass:[NSDictionary class]]) {
    self.caption = captionDictionary[@"text"];
} else {
    self.caption = @"";
}
```

It might seem weird to guess that **`mediaDictionary[@"caption"]`** will return an **`NSDictionary`** and then check if you're correct. If you want, it's possible to instead use **`NSObject`** and cast the result. This will avoid the possibly incorrect assumption entirely, although your code ends u a bit longer:

```
NSObject *captionObject = mediaDictionary[@"caption"];
// Caption might be null (if there's no caption)
if ([captionObject isKindOfClass:[NSDictionary class]]) {
    NSDictionary *captionDictionary = (NSDictionary *)captionObject;
    self.caption = captionDictionary[@"text"];
} else {
    self.caption = @"";
}
```

Again, the difference here is purely stylistic - choose whichever makes the most sense and is easier to read.

Comments

At the end, we iterate through the comments array, pull out the dictionaries one at a time, and pass them to **`BLCComment`** for parsing.

Parsing the Whole Feed

Now that we have the ability to parse different types of objects, we can put it all together to parse the entire Instagram feed when it arrives:

BLCDataSource.m

```
-     NSLog(@"%@", feedDictionary);
+     NSArray *mediaArray = feedDictionary[@"data"];
+
+     NSMutableArray *tmpMediaItems = [NSMutableArray array];
+
+     for (NSDictionary *mediaDictionary in mediaArray) {
+         BLCMedia *mediaItem = [[BLCMedia alloc] initWithDictionary:mediaDictionary];
+
+         if (mediaItem) {
+             [tmpMediaItems addObject:mediaItem];
+         }
+     }
+
+     [self willChangeValueForKey:@"mediaItems"];
+     self.mediaItems = tmpMediaItems;
+     [self didChangeValueForKey:@"mediaItems"];
 }
```

`willChangeValueForKey:` and **`didChangeValueForKey:`** inform the key-value observation system that **`self.mediaItems`** is about to be replaced - and that it has been replaced - by a different array. This will trigger a notification to the table view to reload all of the data.

Downloading Images

When we receive a new **`BLCMedia`** item, we'll want to download the associated image to display it. Let's write a new method to download an image:

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);
+             }
+         });
+     }
+ }
```

This method follows the same pattern as when we connect to the Instagram API. Notably:

1. **`dispatch_async`** to a background queue
2. Make an **`NSURLRequest`**
3. Use **`NSURLConnection`** to connect and get the **`NSData`**
4. Attempt to convert the **`NSData`** into what we're expecting (a **`UIImage`** here)
5. If it works, **`dispatch_async`** back to the main queue, and update our data model with the results

For now, let's start downloading images as they arrive in **`parseDataFromFeedDictionary:fromRequestWithParameters:`**:

BLCDataSource.m

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

This is fairly inefficient, since we're going to start downloading 100 images at roughly the same time. The more appropriate logic, which we'll implement later, starts downloading images as users scroll through the feed.

Table Updates

We have three minor updates to make to our table view and table cells.

Accounting for height calculations with no images

Since **`BLCMedia`** items are no longer guaranteed to have images, we need to update our height constraint logic in **`[BLCMediaTableViewCell -setMediaItem:]`**:

BLCMediaTableViewCell.m

```
- (void) setMediaItem:(BLCMedia *)mediaItem {
    _mediaItem = mediaItem;
    self.mediaImageView.image = _mediaItem.image;
    self.usernameAndCaptionLabel.attributedText = [self usernameAndCaptionString];
    self.commentLabel.attributedText = [self commentString];

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

If we don't check for the presence of an image, when we calculate the constraint, **`self.mediaItem.image.size.width`** will equal zero. It's impossible to divide a number by zero, so the app would crash.

Disabling Cell Selection

We'll have lots of stuff on the screen that looks clickable. For now, let's disable the standard gray background on selection:

BLCMediaTableViewCell.m

```
+ - (void)setHighlighted:(BOOL)highlighted animated:(BOOL)animated {
+     [super setHighlighted:NO animated:animated];
+ }
+
- (void)setSelected:(BOOL)selected animated:(BOOL)animated
{
-     [super setSelected:selected animated:animated];
+     [super setSelected:NO animated:animated];

    // Configure the view for the selected state
}
```

Making rough height estimates

Auto-layout is computationally expensive, and when we add 100 new Instagram posts at a time, it can slow down our UI responsiveness.

By making a rough estimated height, the table view avoids calculating the height of every cell, and instead calculates one at a time as they scroll into the view:

BLCImagesTableViewController.m

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

> *To better understand the effect of this method, comment it out after we implement infinite scroll. You'll notice that without it, the UI freezes briefly while loading more images.*

If you run the app now, you should be able to see images and comments on the initial load, but pull-to-refresh and infinite scroll won't work.

Adding Pull-to-Refresh

We now need to update the data source to support pull-to-refresh for newer images. (Some of these changes also support adding infinite scroll for older images.)

Since we know those methods use a completion handler (**`BLCNewItemCompletionBlock`**), let's start by allowing **`populateDataWithParameters`** to accept a completion block as well.

If there's an error, we'll pass that back, and if not, then we'll pass **`nil`** for the error object:

BLCDataSource.m

```
+ - (void) populateDataWithParameters:(NSDictionary *)parameters completionHandler:(BLCNewItemCompletionBlock)completionHandler {
    if (self.accessToken) {
        // 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_toke

            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];

-                 NSError *jsonError;
-                 NSDictionary *feedDictionary = [NSJSONSerialization JSONObjectWithData:responseData options:0 error:&jsonError];
-
-                 if (feedDictionary) {
+                 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(), ^{
-                         // done networking, go back on the main thread
-                         [self parseDataFromFeedDictionary:feedDictionary fromRequestWithParameters:parameters];
+                         completionHandler(webError);
                    });
                }
            }
        });
    }
}
```

Aside from the error handling and completion blocks, the logic of this method remains the same.

`init` needs to be updated with the new method name:

BLCDataSource.m

```
        // Got a token, populate the initial data
-         [self populateDataWithParameters:nil];
+         [self populateDataWithParameters:nil completionHandler:nil];
```

We'll now update **`requestNewItemsWithCompletionHandler:`** (which is called on pull-to-refresh) to use this method to get newer items. We'll use the **MIN_ID** parameter from the last checkpoint to let Instagram know we're only interested in items with a higher ID (i.e., newer items).

BLCDataSource.m

```
- (void) requestNewItemsWithCompletionHandler:(BLCNewItemCompletionBlock)completionHandler {
    if (self.isRefreshing == NO) {
        self.isRefreshing = YES;

-         // Need to add images here
-
-         self.isRefreshing = NO;
+         NSString *minID = [[self.mediaItems firstObject] idNumber];
+         NSDictionary *parameters = @{@"min_id": minID};

-         if (completionHandler) {
-             completionHandler(nil);
-         }
+         [self populateDataWithParameters:parameters completionHandler:^(NSError *error) {
+             self.isRefreshing = NO;
+
+             if (completionHandler) {
+                 completionHandler(error);
+             }
+         }];
    }
}
```

Currently, **`populateDataWithParameters:completionHandler:`** calls **`parseDataFromFeedDictionary:fromRequestWithParameters:`**, which ignores the **`parameters`** dictionary. Let's update this method to inspect it, and place the newly parsed media items appropriately:

BLCDataSource.m

```
-     [self willChangeValueForKey:@"mediaItems"];
-     self.mediaItems = tmpMediaItems;
-     [self didChangeValueForKey:@"mediaItems"];
+     NSMutableArray *mutableArrayWithKVO = [self mutableArrayValueForKey:@"mediaItems"];
+
+     if (parameters[@"min_id"]) {
+         // This was a pull-to-refresh request
+
+         NSRange rangeOfIndexes = NSMakeRange(0, tmpMediaItems.count);
+         NSIndexSet *indexSetOfNewObjects = [NSIndexSet indexSetWithIndexesInRange:rangeOfIndexes];
+
+         [mutableArrayWithKVO insertObjects:tmpMediaItems atIndexes:indexSetOfNewObjects];
+     } else {
+         [self willChangeValueForKey:@"mediaItems"];
+         self.mediaItems = tmpMediaItems;
+         [self didChangeValueForKey:@"mediaItems"];
+     }
```

Pull-to-refresh will now work, and we're set up for success with infinite scroll. Let's take care of that.

Adding Infinite Scroll

Sometimes you can infinite scroll, but there are no more older messages. We'll add a new property - **`self.thereAreNoMoreOlderMessages`** - t prevent pointless requests:

BLCDataSource.m

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

We'll update **`requestOldItemsWithCompletionHandler:`** to check this property and use the **MAX_ID** Instagram API key:

```
- (void) requestOldItemsWithCompletionHandler:(BLCNewItemCompletionBlock)completionHandler {
-     if (self.isLoadingOlderItems == NO) {
+     if (self.isLoadingOlderItems == NO && self.thereAreNoMoreOlderMessages == NO) {
        self.isLoadingOlderItems = YES;

-         // Need to add images here
+         NSString *maxID = [[self.mediaItems lastObject] idNumber];
+         NSDictionary *parameters = @{@"max_id": maxID};

-         self.isLoadingOlderItems = NO;
-
-         if (completionHandler) {
-             completionHandler(nil);
-         }
+         [self populateDataWithParameters:parameters completionHandler:^(NSError *error) {
+             self.isLoadingOlderItems = NO;
+
+             if (completionHandler) {
+                 completionHandler(error);
+             }
+         }];
    }
 }
```

Let's also reset **`thereAreNoMoreOlderMessages`** if the user does a pull-to-refresh:

BLCDataSource.m

```
 - (void) requestNewItemsWithCompletionHandler:(BLCNewItemCompletionBlock)completionHandler {
+     self.thereAreNoMoreOlderMessages = NO;
```

Finally, we'll have **`parseDataFromFeedDictionary:fromRequestWithParameters:`** check for **`max_id`** in **`parameters`**:

BLCDataSource.m

```
    if (parameters[@"min_id"]) {
        // This was a pull-to-refresh request

        NSRange rangeOfIndexes = NSMakeRange(0, tmpMediaItems.count);
        NSIndexSet *indexSetOfNewObjects = [NSIndexSet indexSetWithIndexesInRange:rangeOfIndexes];

        [mutableArrayWithKVO insertObjects:tmpMediaItems atIndexes:indexSetOfNewObjects];
+     } else if (parameters[@"max_id"]) {
+         // This was an infinite scroll request
+
+         if (tmpMediaItems.count == 0) {
+             // disable infinite scroll, since there are no more older messages
+             self.thereAreNoMoreOlderMessages = YES;
+         }
+
+         [mutableArrayWithKVO addObjectsFromArray:tmpMediaItems];
    } else {
        [self willChangeValueForKey:@"mediaItems"];
        self.mediaItems = tmpMediaItems;
        [self didChangeValueForKey:@"mediaItems"];
    }
```

Now you can scroll to the bottom, and your Instagram client will try to load older messages.

Recap

Congratulations! You now have a basic Instagram client with these features:

- Pull-to-refresh to check for newer images
- Load older images when scrolling to the bottom (infinite scroll)

You've also learned how to download data from the Internet and serialize it on a background queue. This is important stuff - give yourself a pat on the back!

Before you start your assignment, commit, merge and push your code to GitHub:

Terminal

```
$ git add .
$ git commit -m 'Add Instagram Images'
$ git checkout master
$ git merge instagram-images
$ git push
```

Then, check out a new branch for this assignment:

Terminal

```
$ git checkout -b instagram-images-bugfix
```

There's a minor bug in our app: if you swipe-to-delete all of the images, and then do pull-to-refresh, you'll cause a crash here:

```
- (void) requestNewItemsWithCompletionHandler:(BLCNewItemCompletionBlock)completionHandler {
    self.thereAreNoMoreOlderMessages = NO; // restart infinite scroll after a pull-to-refresh

    if (self.isRefreshing == NO) {
        self.isRefreshing = YES;

        NSString *minID = [[self.mediaItems firstObject] idNumber];
        NSDictionary *parameters = @{@"min_id": minID};

        [self populateDataWithParameters:parameters completionHandler:^(NSError *error) {
            self.isRefreshing = NO;

            if (completionHandler) {
                completionHandler(error);
            }
        }];
    }
}
```

Try to figure out why this happens and fix it.

Push your progress to GitHub *without* merging it into master:

Terminal

```
$ git push -u origin instagram-images-bugfix
```

If you were able to fix it, submit a pull request to your mentor with your proposed fix.

If you were not, send a message to your mentor with your hypothesis (or hypotheses) about why it crashes here, and a description of the efforts you made to avoid the crash.

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

Remembering Images and Login between Launches

> *"Geez! Nice pod design, Frank Lloyd Wrong."*
>
> *- Captain Hazel "Hank" Murphy, Fictional Character*

You may have noticed: every time we run the app it starts on the Instagram login web view.

Ideally, we'd remember the user between app launches, and bring them to where they last left off.

In this checkpoint, you'll learn about common methods of **persistence** in iOS.

Introducing the Keychain

The most important thing for us to remember right now is the access token we receive from the Instagram API. We can always re-download images and comments, but if we lose the access token, we won't be able to do anything unless the user logs in again.

The access token isn't just important: it's also important to keep *secure*. This means we need to prevent unauthorized access to the token.

iOS makes it relatively simple to encrypt data and store  rely. Access tokens (and other confidential credentials, like usernames, e-mail addresses, certificates, and passwords) can be stored in *ychain*. The keychain is encrypted by the user's passcode using **AES-256 encryption**. It would be very difficult to break, especially if the user has a complex passcode.

Apple's **Keychain Services Reference** and **Keychain Services Programming Guide** show that the Keychain API has a lot of flexibility and power.

With great power comes great amounts of complexity. The Keychain API is an advanced topic that you'll explore as you gain experience. Instead of teaching this API, we'll show you how to find third-party libraries that simplify common problems and install them in your app.

Ultimately, we'll install `UICKeyChainStore`, a nice, open-source keychain library by iOS developer Kishikawa Katsumi.

While we could **download the files** and copy them into our app, a more future-proof solution is to install a tool called **CocoaPods**, which allows for simple installation and updates of third-party tools.

Installing CocoaPods

Quit Xcode.

Open up Terminal and run this command:

Terminal

```
$ sudo gem install cocoapods --verbose
```

> **`sudo`** *tells the command line prompt to run the following command with administrator privileges.*
>
> **`gem`** *is a tool that installs software written in the programming language* ***Ruby*** *(which CocoaPods is).*
>
> **`--verbose`** *is optional; it tells* **`gem`** *to be verbose in describing what's happening. You don't need it; it's just more fun to watch.*

At the end, you should see a message like **`18 gems installed`** (the number may vary from system to system).

CocoaPods is now installed on your Mac. We'll now configure Blocstagram to use CocoaPods.

Configuring CocoaPods

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

Terminal

```
$ git checkout -b install-cocoapods
```

Now, we'll write a **Podfile**. A Podfile is a list of third-party libraries that you want embedded in your app. There's a searchable list on **cocoapods.org**.

Create a blank Podfile and open it in your favorite text editor.

Terminal

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

Add this line:

Podfile

```
pod 'UICKeyChainStore'
```

Save and close the window, and return to the command line:

Terminal

```
$ pod install
```

The eventual output should look like this:

Terminal

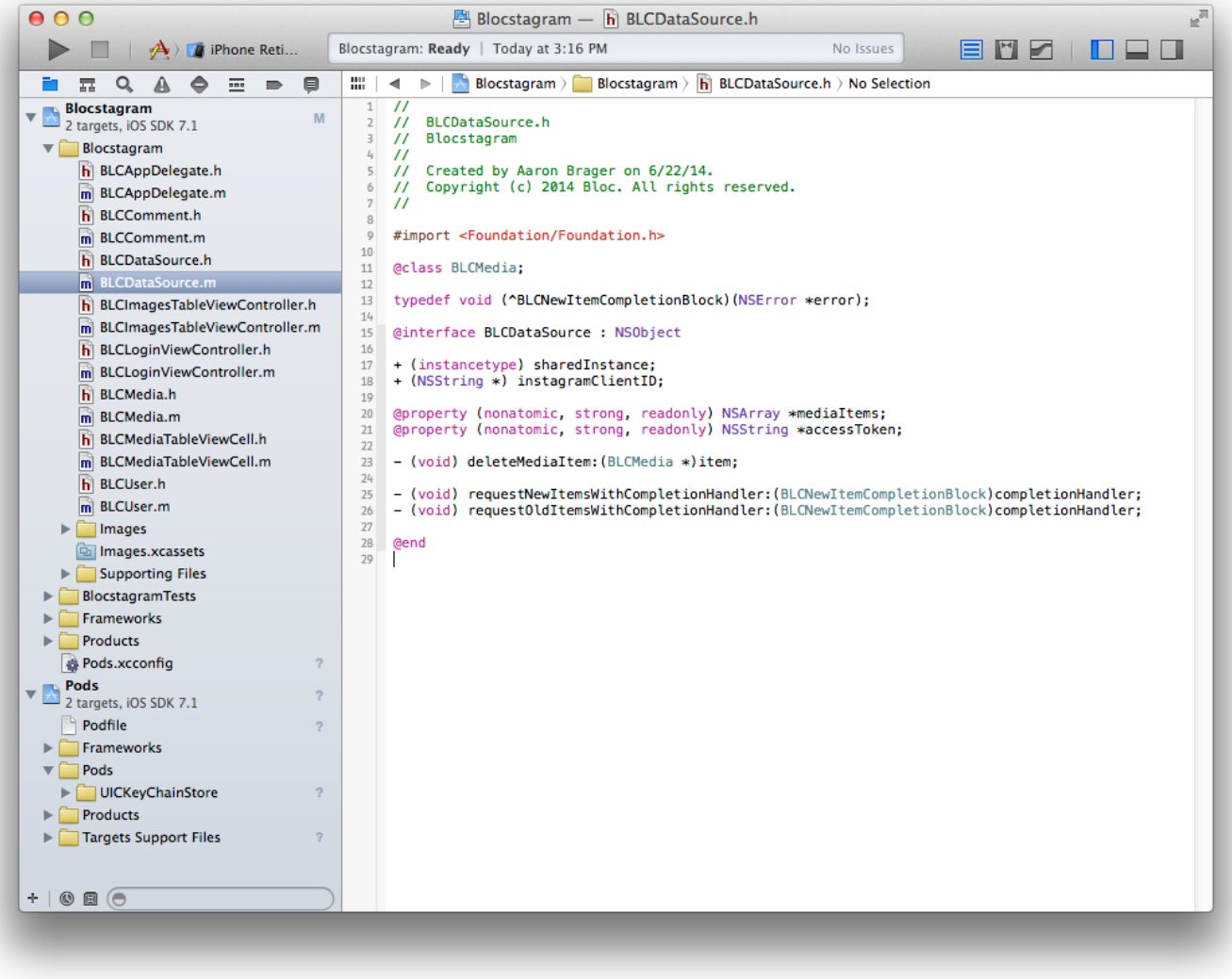
```
Downloading dependencies
Installing UICKeyChainStore (1.0.5)
Generating Pods project
Integrating client project

[!] From now on use `Blocstagram.xcworkspace`.
```

You'll now have a bunch of new files in your Blocstagram folder. Open it up in Finder and double-click on **`Blocstagram.xcworkspace`** (**not `Blocstagram.xcodeproj`**).

> *An* ***Xcode Workspace*** *groups together multiple Xcode projects so they can be used together*

If everything went smoothly, you should now be able to see your Blocstagram project and your Pods together in the Project navigator:



We've accomplished two important things:

- CocoaPods is configured, so it'll be easy to install other third-party libraries in the future
- We've installed **`UICKeyChainStore`**, allowing us easy access to the iOS keychain.

Commit your changes and push them:

Terminal

```
$ git add .
$ git commit -m 'Configured CocoaPods'
$ git checkout master
$ git merge install-cocoapods
$ git push
```

Make a new **`git`** branch for our functional changes:

Terminal

```
$ git checkout -b persist-data
```

Let's set up the keychain.

Using the Keychain

In BLCDataSource, we'll check for and save the access token.

First, import the new library:

BLCDataSource.m

```
#import "BLCMedia.h"
#import "BLCComment.h"
#import "BLCLoginViewController.h"
+ #import <UICKeyChainStore.h>
```

> *Use quotes (" ") when importing local files; use angle brackets (< >) when importing external files.*

In **registerForAccessToken**, save the token:

BLCDataSource.m

```
- (void) registerForAccessTokenNotification {
    [[NSNotificationCenter defaultCenter] addObserverForName:BLCLoginViewControllerDidGetAccessTokenNotification object:nil queue:nil
        self.accessToken = note.object;
+         [UICKeyChainStore setString:self.accessToken forKey:@"access token"];
```

And in **init**, check for the token:

BLCDataSource.m

```
    if (self) {
+         self.accessToken = [UICKeyChainStore stringForKey:@"access token"];
+
+         if (!self.accessToken) {
-         [self registerForAccessTokenNotification];
+             [self registerForAccessTokenNotification];
+         } else {
+             [self populateDataWithParameters:nil completionHandler:nil];
+         }
    }
```

Now the token will be saved and restored.

We'll now update our app delegate to show the appropriate view upon app launch:

BLCAppDelegate.m

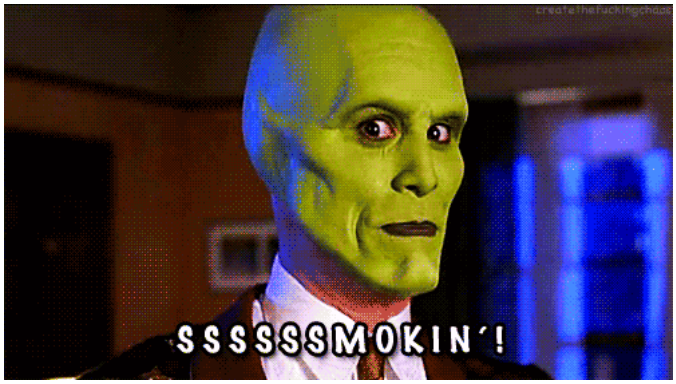
```
    UINavigationController *navVC = [[UINavigationController alloc] init];
+
+     if (![BLCDataSource sharedInstance].accessToken) {
        BLCLoginViewController *loginVC = [[BLCLoginViewController alloc] init];
        [navVC setViewControllers:@[loginVC] animated:YES];


        [[NSNotificationCenter defaultCenter] addObserverForName:BLCLoginViewControllerDidGetAccessTokenNotification object:nil queue
            BLCImagesTableViewController *imagesVC = [[BLCImagesTableViewController alloc] init];
            [navVC setViewControllers:@[imagesVC] animated:YES];
        }];
+     } else {
+         BLCImagesTableViewController *imagesVC = [[BLCImagesTableViewController alloc] init];
+         [navVC setViewControllers:@[imagesVC] animated:YES];
+     }
```

Test it:

- Login to Instagram
- Quit the app
- Run it again

You no longer need to login!



> *If you're curious, feel free to read through the* **`UICKeyChainStore.m`** *source code to see how it works.*

Saving Instagram Data to Disk

The keychain is only good for storing small amounts of privacy-critical data. We can use **`NSCoding`** and **`NSKeyedArchiver`** together to write all of the Instagram data to the disk, and then read it when the app launches.

`NSCoding` is a *protocol* like **`UITableViewDelegate`**. Objects *conform* to it when they implement the specific methods and properties.

`NSKeyedArchiver` is an object that converts between saved files and objects that conform to **`NSCoding`**.

In other words, if our objects *conform* to **`NSCoding`**, we can save them to disk.

Conforming to **`NSCoding`**

According to the **`NSCoding` Protocol Reference** we'll need to implement two methods in the objects we want to save:

1. **`encodeWithCoder:`**. We are given an **`NSCoder`** object, and we save relevant data to it. (This is later written to disk.)
2. **`initWithCoder:`**. The **`NSCoder`** object has been read from disk, and we turn it back into an object.

An example will help. Here's how it looks in **`BLCUser`**:

Declare that we conform to the protocol:

BLCUser.h

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

Add **`encodeWithCoder:`**:

BLCUser.m

```
+
+ - (void) encodeWithCoder:(NSCoder *)aCoder {
+     [aCoder encodeObject:self.idNumber forKey:NSStringFromSelector(@selector(idNumber))];
+     [aCoder encodeObject:self.userName forKey:NSStringFromSelector(@selector(userName))];
+     [aCoder encodeObject:self.fullName forKey:NSStringFromSelector(@selector(fullName))];
+     [aCoder encodeObject:self.profilePicture forKey:NSStringFromSelector(@selector(profilePicture))];
+     [aCoder encodeObject:self.profilePictureURL forKey:NSStringFromSelector(@selector(profilePictureURL))];
+ }
```

You'll notice a new function: **`NSStringFromSelector`**. Here we're converting our selectors (**`idNumber`**, **`userName`**, etc.) into strings.

Instead of writing this:

```
[aCoder encodeObject:self.idNumber forKey:NSStringFromSelector(@selector(idNumber))];
```

We could write this:

```
[aCoder encodeObject:self.idNumber forKey:@"idNumber"];
```

The former is more annoying to look at, but will give you a warning if you make a typo. The latter is simpler to read, but can be an annoying source of hard-to-fix bugs.

Now add **`initWithCoder:`**:

BLCUser.m

```
 #pragma mark - NSCoding
+
+ - (instancetype) initWithCoder:(NSCoder *)aDecoder {
+     self = [super init];
+
+     if (self) {
+         self.idNumber = [aDecoder decodeObjectForKey:NSStringFromSelector(@selector(idNumber))];
+         self.userName = [aDecoder decodeObjectForKey:NSStringFromSelector(@selector(userName))];
+         self.fullName = [aDecoder decodeObjectForKey:NSStringFromSelector(@selector(fullName))];
+         self.profilePicture = [aDecoder decodeObjectForKey:NSStringFromSelector(@selector(profilePicture))];
+         self.profilePictureURL = [aDecoder decodeObjectForKey:NSStringFromSelector(@selector(profilePictureURL))];
+     }
+
+     return self;
+ }

 - (void) encodeWithCoder:(NSCoder *)aCoder {
```

That's it. **`BLCUser`** now conforms to **`NSCoding`**.

Now, let's add nearly identical code to **`BLCComment`** and **`BLCMedia`**:

BLCComment.h

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

BLCComment.m