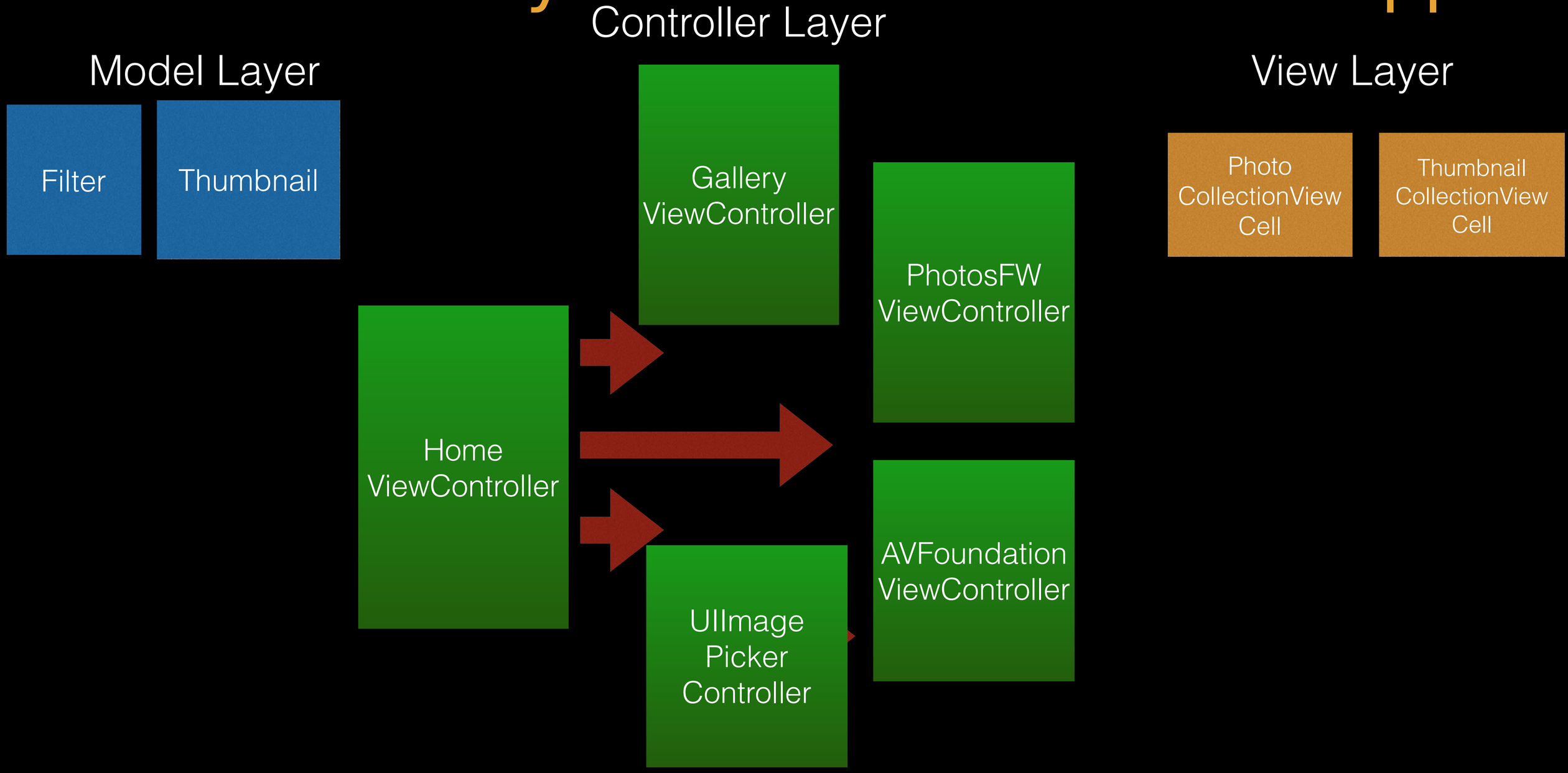
# iOS Dev Accelerator Week2 Day1

- Programmatic Layout
- UIAlertController
- Collection Views
- Asset Catalogs
- UIImagePickerController

# The MVC layout of our Week 2 App



# Programmatic Layout

# Programmatic Layout

- Sometimes, laying out your interface is more convenient to do in code rather than in a storyboard or nib.
- This will add a lot more code to your app, but also gives you more control over your interface.
- For this weeks app, we wont be using any storyboards or nibs!

# No storyboard

- In order to not use the storyboard we need to do 2 things:

    1. Delete the storyboard entry in our Project's settings

    2. Change our App delegate to to programmatically create a root view controller:

```
class AppDelegate: UIResponder, UIApplicationDelegate {

    var window: UIWindow?


    func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions:
            [NSObject: AnyObject]?) -> Bool {
        // Override point for customization after application launch.
        self.window = UIWindow(frame: UIScreen.mainScreen().bounds)
        self.window?.makeKeyAndVisible()
        var rootVC = ViewController()
        self.window?.rootViewController = rootVC
        return true
    }
```

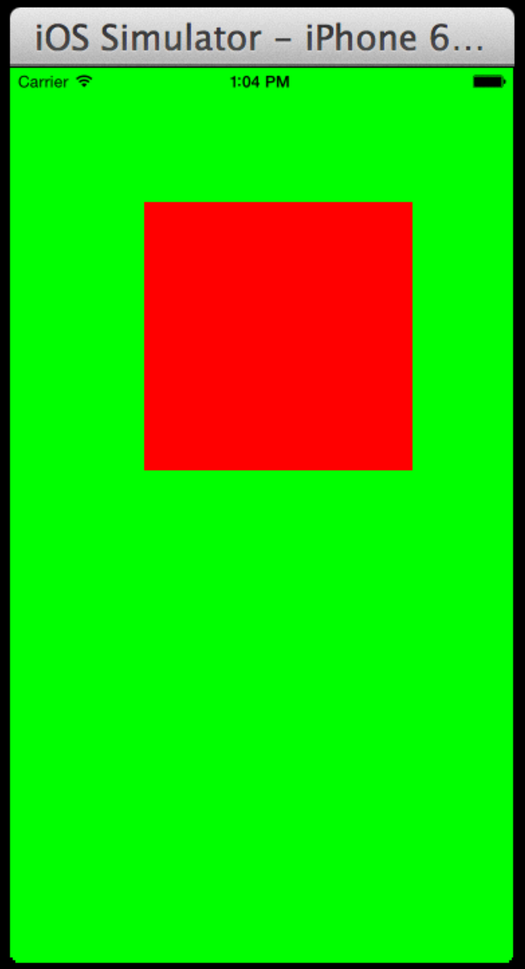
# loadView

- When you are creating your views programmatically, the proper place to do it is by overriding your view controller's loadView method. loadView is called even before viewDidLoad!
- The implementation of this method should follow these guidelines:
    1. Create a root UIView object. This view will contain all other views for this view controller. You usually instantiate this view with a frame that matches the size of the app's window, which fills the screen. You don't technically need to do this though, if you don't give it a frame it will just be full screen.
    2. Create additional subviews and add them to the root view
    3. Add any autolayout constraints you need to your view hierarchy.
    4. Assign the root view to the view property of your view controller

*this workflow sourced from http://matthewmorey.com/creating-uiviews-programmatically-with-auto-layout/

# loadView example

```
class ViewController: UIViewController {


    override func loadView() {
        let rootView = UIView(frame: UIScreen.mainScreen().
       bounds)
        rootView.backgroundColor = UIColor.greenColor()
        let redBox = UIView(frame: CGRect(x: 100, y: 100,
       width: 200, height: 200))
        redBox.backgroundColor = UIColor.redColor()
        rootView.addSubview(redBox)
        self.view = rootView
    }
```

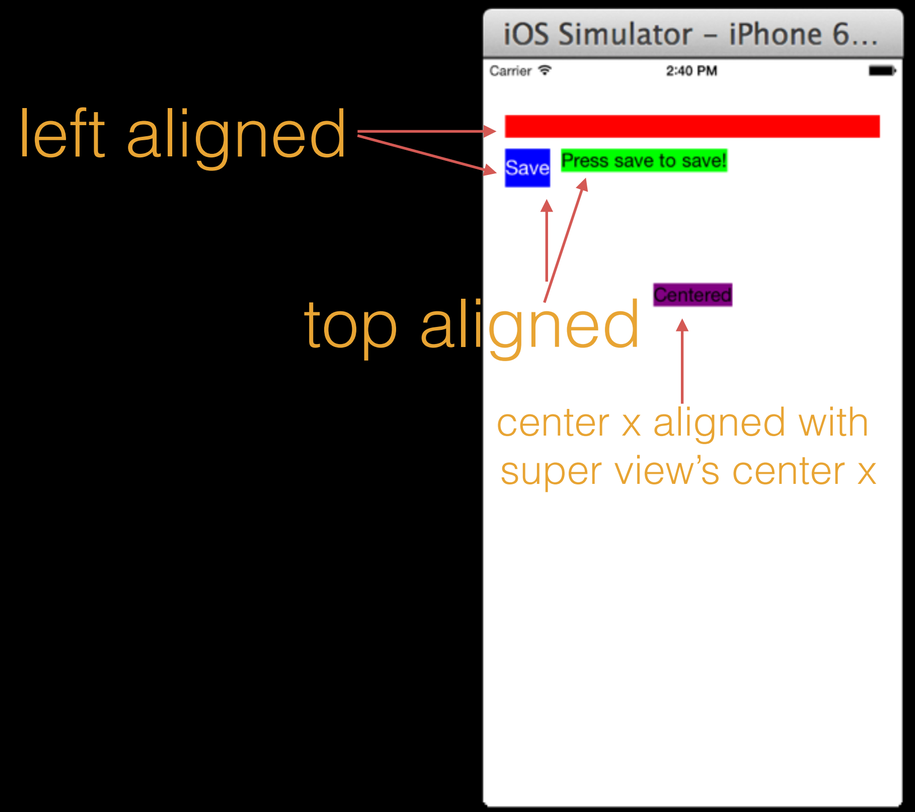


# Programmatic Autolayout

- You can achieve all the same things with autolayout by doing it in code vs the storyboard.
- These are the 3 ways to create autolayout constraints, ranked in order of Apples recommendation :
    1. Storyboard/Nib
    2. Creating Constraints with Visual Format Language in code
    3. Creating Constraint objects in code
- All 3 can be mixed and matched together, as we will see in a bit

# Our Mission

- To achieve this stunning layout without storyboard or nib:



# VFL (Visual Format language)

- Constraints are represented by instances of the class NSLayoutConstraint
- To create constraints with VFL, you use the class method constraintWithVisualFormat:options:metrics:views:
- The first parameter of the method is a visual format string. This is just a string formatted in a specific way that describes the constraints you want. Lets take a look at how these things work

# VFL (Visual Format language)

Standard Space

```
[button]-[textField]
```


Connection to Superview

```
|-50-[purpleBox]-50-|
```


Multiple Predicates

```
[flexibleButton(>=70,<=100)]
```


Vertical Layout

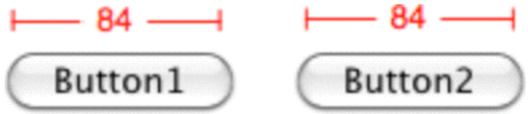
```
V:[topField]-10-[bottomField]
```

Flush Views

```
[maroonView][blueView]
```

Equal Widths

```
[button1(==button2)]
```



# VFL (Visual Format language)

- The first step is to create the views that are going to makeup our interface, and add them to the root view.
- Notice how we didn't give any of them frames. Autolayout will end up doing this for us

```
let redView = UIView()
redView.backgroundColor = UIColor.redColor()
let blueButton = UIButton()
blueButton.setTitle("Save", forState: UIControlState.Normal)
blueButton.backgroundColor = UIColor.blueColor()
let greenLabel = UILabel()
greenLabel.text = "Press save to save!"
greenLabel.backgroundColor = UIColor.greenColor()
let pinkLabel = UILabel()
pinkLabel.text = "Centered"
pinkLabel.backgroundColor = UIColor.purpleColor()

rootView.addSubview(redView)
rootView.addSubview(blueButton)
rootView.addSubview(greenLabel)
rootView.addSubview(pinkLabel)
```

# VFL (Visual Format language)

- Next step is to set translatesAutosizingMaskIntoConstraints to false
- And then we create a dictionary that will contain key value pairings for all the views you are going to be applying constraints to

```
redView.setTranslatesAutoresizingMaskIntoConstraints(false)
blueButton.setTranslatesAutoresizingMaskIntoConstraints(false)
greenLabel.setTranslatesAutoresizingMaskIntoConstraints(false)
pinkLabel.setTranslatesAutoresizingMaskIntoConstraints(false)

let views = ["redView" : redView,
             "blueButton" : blueButton,
             "greenLabel" : greenLabel,
              "pinkLabel" : pinkLabel]
```

# VFL (Visual Format language)

- It's finally time to start making constraints!
- When writing your visual format strings, you begin with either H (Horizontal) or V (Vertical). This will dictate if the constraints are applied to the horizontal dimension (leading/trailing) or vertical dimension (top/bottom).
- Every view in your visual format string is represented in square brackets.
- The names of the views come from that dictionary we setup. The dictionary is actually passed in as parameter.
- Connections between views is represented by using a hyphen, or two hyphens with a number in between to represent the number of points the views should be.
- The super view is represented by | pipes
- You can use parens () to set values for fixed Width and Height
- Lets look at some examples in code

# VFL (Visual Format language)

```
let redViewContraintsHeight = NSLayoutConstraint.constraintsWithVisualFormat(
    "V:[redView(20)]",
    options: nil,
    metrics: nil,
    views: views)
redView.addConstraints(redViewContraintsHeight)
```

- Here we are constraining the height of the redView.
- We add the constraint to the redView itself. A rule of thumb is anytime you are containing the size of the view, you add the constraints to the view itself. If you are constraining the position, you add the constraints to it's super view.

# VFL (Visual Format language)

```
let redViewConstraintY = NSLayoutConstraint.constraintsWithVisualFormat(
    "V:|-50-[redView]",
    options: nil,
    metrics: nil,
    views: views)
rootView.addConstraints(redViewConstraintY)
let redViewConstraintX = NSLayoutConstraint.constraintsWithVisualFormat(
    "H:|-20-[redView]-20-|",
    options: nil,
    metrics: nil,
    views: views)
rootView.addConstraints(redViewConstraintX)
```

- Next we constrain the the position of the redView
- This time we add the constraints to the rootView, which is redView's super view.

# Demo

# Centering With SuperView

- For whatever reason, Apple made it terribly difficult to use VFL to align a view's centerX or Y with its superview's centerX or Y
- So instead of using VFL for that, we can use the 3rd way of using autolayout, creating constraint objects in code.
- It works pretty much like you would expect. It takes in parameters for each part of the constraint equation.
- Heres that equation again:
  - first attribute = second attribute * multiplier + constant

# Centering With SuperView

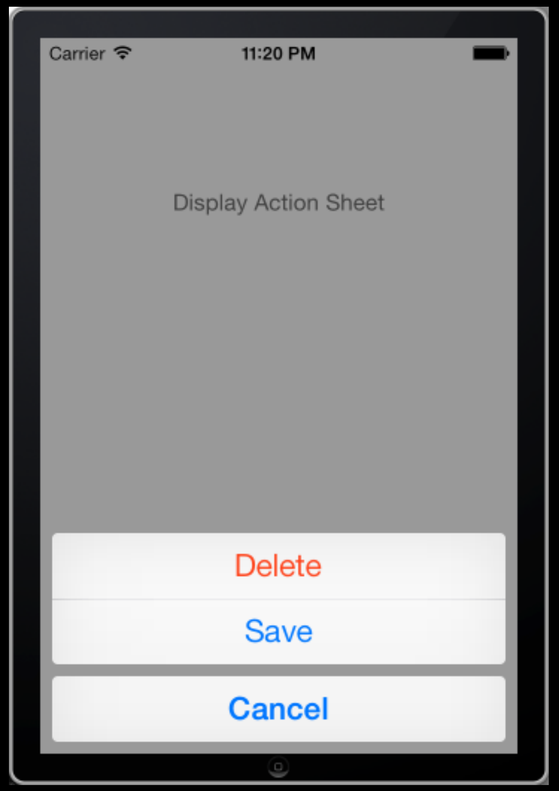
```
let pinkLabelCenterX = NSLayoutConstraint(item: pinkLabel,
                                          attribute: .CenterX,
                                          relatedBy:.Equal,
                                             toItem: rootView,
                                          attribute: .CenterX,
                                         multiplier: 1.0,
                                           constant: 0.0)

rootView.addConstraint(pinkLabelCenterX)
```

# Demo

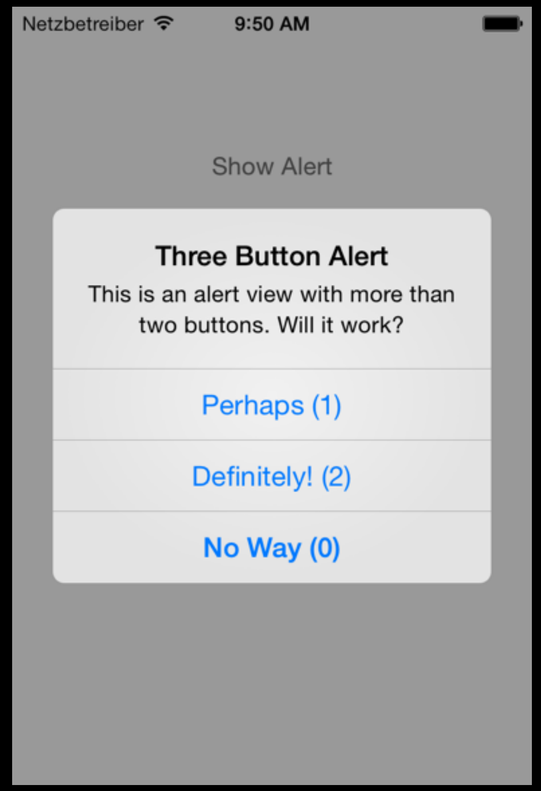
# UIAlertController

# UIAlertController

- “UIAlertController object displays an alert message to the user”
- Replaces both UIActionSheet and UIAlertView in iOS8
- After configuring the Alert Controller present it with presentViewController:animated: Completion:




ActionSheet




AlertView

# UIAlertController Setup

`init(title:message:preferredStyle:)`

Creates and returns a view controller for displaying an alert to the user.

**Declaration**

SWIFT

```
convenience init(title title: String!,
                 message message: String!,
           preferredStyle preferredStyle: UIAlertControllerStyle)
```

**Parameters**

| | |
|---|---|
| *title* | The title of the alert. Use this string to get the user's attention and communicate the reason for the alert. |
| *message* | Descriptive text that provides additional details about the reason for the alert. |
| *preferredStyle* | The style to use when presenting the alert controller. Use this parameter to configure the alert controller as an action sheet or as a modal alert. |

# UIAlertController Configuration

- In order to add buttons to your alert controller, you need to add actions.
- An action is a instance of the UIAlertAction class.
- “A UIAlertAction object represents an action that can be taken when tapping a button in an alert”
- Uses a closure expression (great!) to define the behavior of when the button is pressed. This is called the handler.

# UIAlertAction Setup

`init(title:style:handler:)`

Create and return an action with the specified title and behavior.

**Declaration**

SWIFT

```
convenience init(title title: String!,
                 style style: UIAlertActionStyle,
               handler handler: ((UIAlertAction!) -> Void)!)
```

**Parameters**

| | |
|---|---|
| *title* | The text to use for the button title. The value you specify should be localized for the user's current language. This parameter must not be `nil`. |
| *style* | Additional styling information to apply to the button. Use the style information to convey the type of action that is performed by the button. For a list of possible values, see the constants in UIAlertActionStyle. |
| *handler* | A block to execute when the user selects the action. This block has no return value and takes the selected action object as its only parameter. |

**Return Value**

A new alert action object.

Feedback

# Adding Actions

- Adding actions to the AlertController is as easy as calling addAction: on your AlertController and passing in the UIAlertAction(s)
- The order in which you add those actions determines their order in the resulting AlertController.

# Presenting the alert controller

- To present the alert controller, you can call presentViewController:animated:completion: on the parent view controller
- This will work out of the box for iPhone, but on iPad it takes a bit more configuration
- On iPad you have to tell the alert controller where to present from, since its going basically be a pop out menu.
- You can do this by setting the sourceView and sourceRect on the alert controller's popoverPresentationController.
- There is a weird bug with this, where the Alert controllers popoverPresentationController gets reset every time you present it, so you have to configure it every time before you are about to present.
- We will do this in the demo.

# Demo

# UICollectionView

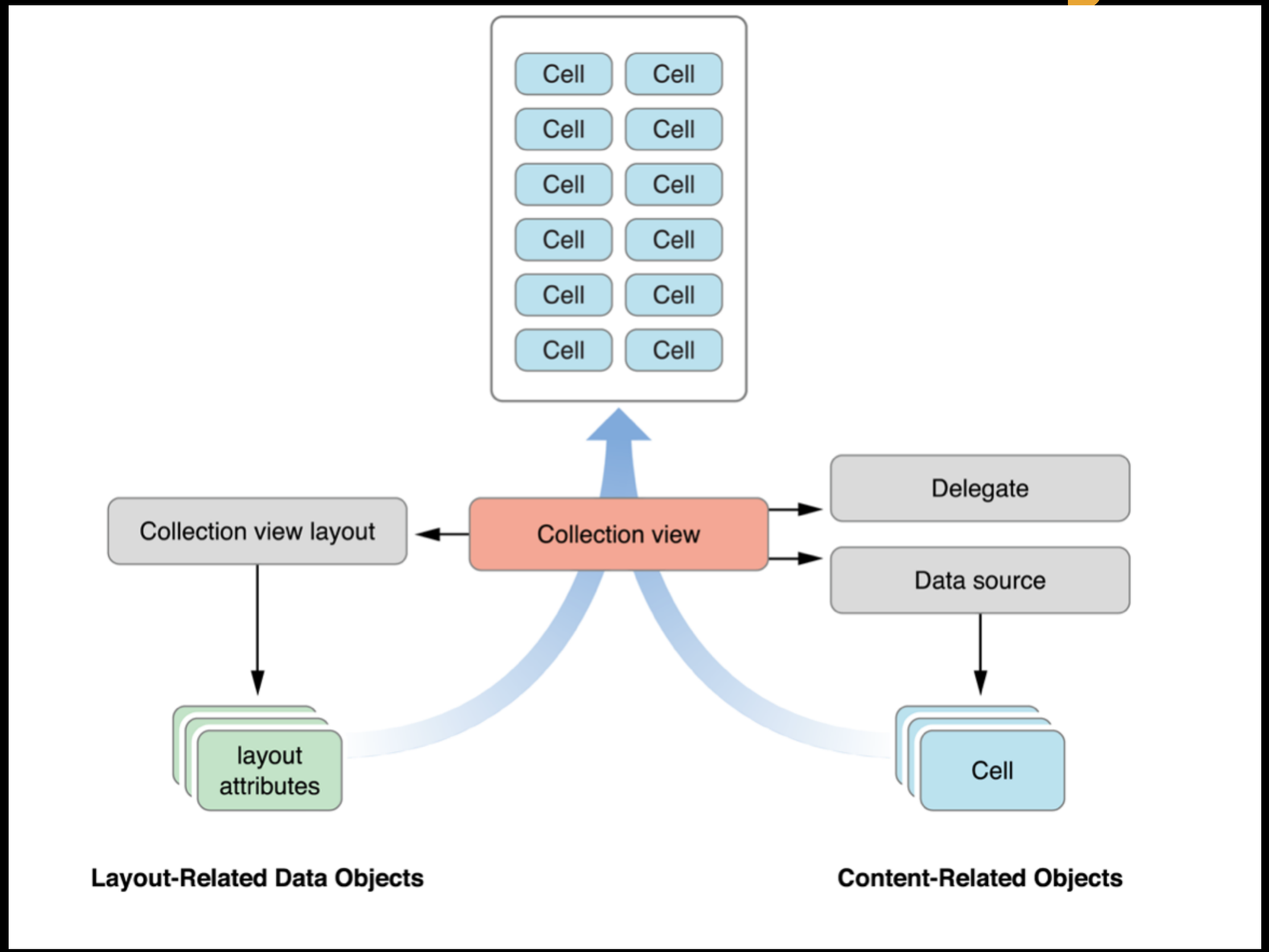
# UICollectionView

- “A collection view is a way to present an ordered set of data items using flexible and changeable layout”
- Most commonly used to present items in a grid-like arrangement. (Items to collection views are as rows to table views)
- Creating custom layouts allow the possibility of many different layouts (grids, circular layouts, stacks, dynamic,etc)

# Internal WorkFlow

- You provide the data (datasource pattern!)
- The layout object provides the placement information
- The collection view merges the two pieces together to achieve the final appearance.

# Collection View Objects



# Reusable Views

- Collection views employ the same recycle program that table views do. (Same Queue data structure)
- 3 reusable views involved with collection views:
    1. Cells : Presents the content of a single item
    2. Supplementary views : headers and footers
    3. Decoration views : wholly owned by the layout object and not tied to any data from your data source. Ex : Custom background appearance.
- Unlike table views, collection view imposes no styles on your reusable views, they are for the most part blank canvases for you to work with.

# CollectionViewDataSource

- Very similar to tableview's datasource. It is required.
- Must answer these questions:
  - For a given section, how many items does a section contain?
  - For a given section or item, what views should be used to display the corresponding content? (just like cellForRow)

# CollectionViewLayout

- Since we are not using storyboard, we need to programmatically adjust the look of our collection view by accessing its layout property.
- We are going to use the prebuilt layout provided by apple, which is called ‘Flow Layout’
- All we are going to use it for initially is to set the size of our cells. Normally you can do this in storyboard.

# Demo

# Asset Catalogs

# Asset Catalogs

- “Use asset catalogs to simplify management of images that are used by your app”
- Things you can put in your asset catalogs:
  - Image sets
  - App icons
  - Launch Images

# Image Sets

- Image sets contain all the versions of an image necessary for the different scale factors of different devices' screens. (ie retina vs non-retina)
- When you drag an image into the XCAsset, an image set will be created for it.
- You will see 1x(non retina), 2x (retina), and 3x (retina HD) slots for each image.
- Image sets can also be configured to be device or size class specific.

# 1x, 2x, and 3x

- Lets say we have an image view that is going to be constrained to 300 points height and 300 points width.
- We would need a 300x300 image for 1x, 600x600 image for 2x, and 900x900 image for 3x for the image displayed in the image view to scale properly for each device resolution.

# 1x, 2x, and 3x

# 1x, 2x, and 3x

# Demo

# Camera Programming

- 2 ways for interfacing with the camera in your app:
  1. UIImagePickerController (easy mode)
  2. AVFoundation Framework (hard mode)

# UIImagePickerController

- The workflow of using UIImagePickerController is 3 steps:
    1. Instantiate and modally present the UIImagePickerController
    2. ImagePicker manages the user's interaction with the camera or photo library
    3. The system invokes your image picker controller delegate methods to handle the user being done with the picker.

# UIImagePickerController Setup

- The first thing you have to account for is checking if the device has a camera.
- If your app absolutely relies on a camera, add a UIRequiredDeviceCapabilities key in your info.plist
- Use the isSourceTypeAvailable class method on UIImagePickerController to check if camera is available.

# UIImagePickerController Setup

- Next make sure something is setup to be the delegate of the picker. This is usually the view controller that is spawning the picker.
- The final step is to actually create the UIImagePicker with a sourceType of UIImagePickerControllerSourceTypeCamera.
- Media Types: Used to specify if the camera should be locked to photos, videos, or both.
- AllowsEditing property to set if the user is able to modify the photo in the picker after taking the photo.

# UIImagePickerControllerDelegate

- The Delegate methods control what happens after the user is done using the picker. 2 big method:
  1. imagePickerControllerDidCancel:
  2. imagePickerController:didFinishPickingMediaWithInfo:

# Info Dictionary

The info dictionary has a number of items related to the image that was taken:

```
NSString *const UIImagePickerControllerMediaType;
NSString *const UIImagePickerControllerOriginalImage;
NSString *const UIImagePickerControllerEditedImage;
NSString *const UIImagePickerControllerCropRect;
NSString *const UIImagePickerControllerMediaURL;
NSString *const UIImagePickerControllerReferenceURL;
NSString *const UIImagePickerControllerMediaMetadata;
```

MediaType is either kUTTypeImage or kUTTypeMovie

# Demo