lecture 10

UI/UX and Programmatic Design

cs198-001 : fall 2017

Announcements

Custom App progress form due before lab (~1 minute)

Only 2 labs left (no lab next week)

Final Presentations: Friday of RRR at 3:30pm in HP Auditorium

- If you have a conflict, let us know on Piazza now

Wednesday's Lab

Markdown mini-lecture (how to style GitHub README files)

Meet with your assigned TA and go over your progress so far / discuss things you may need to change

iOS Human Interface Guidelines

Your “go-to” resource for best practices concerning correct usage / placement / properties of UI elements (link)

iOS Human Interface Guidelines

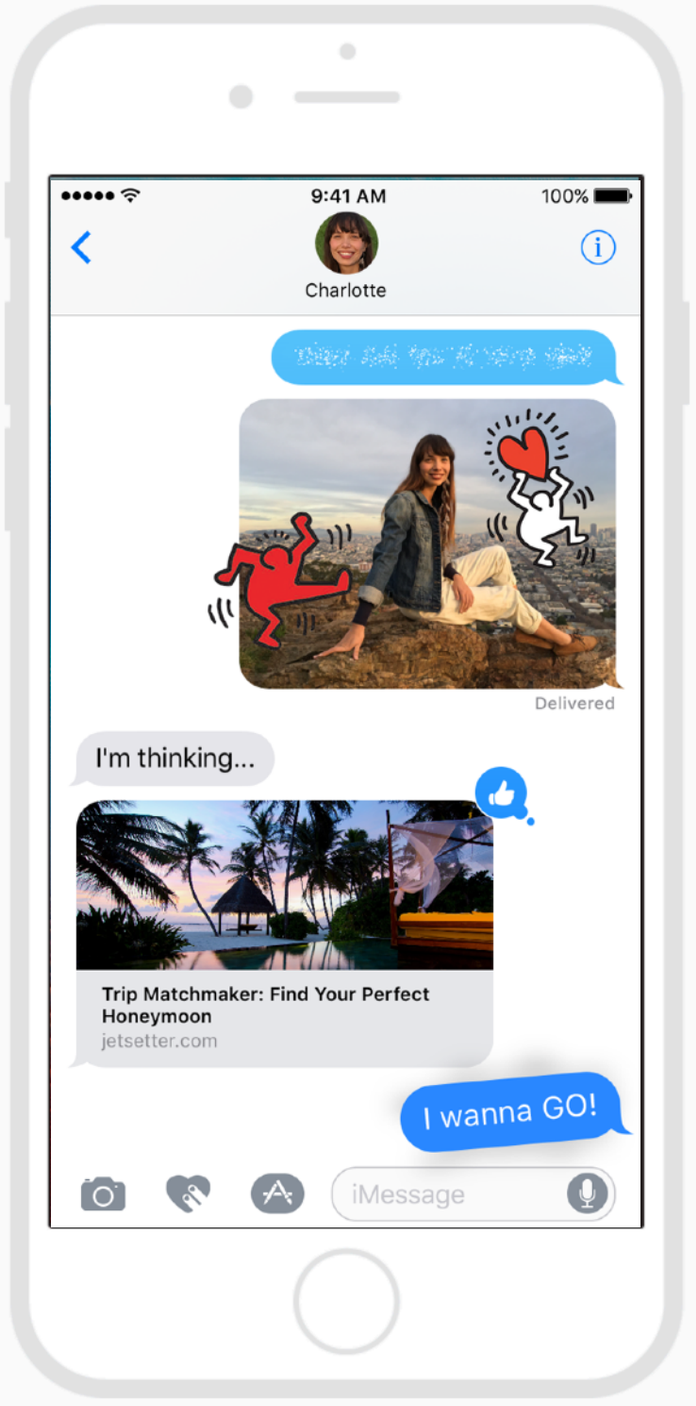
iMessage Integration

Enables you to implement a Messaging Extension for your app

Can share text, photo, stickers, interactive games (in-message!)

For iMessage Apps, be sure to have a distinct focus (should be relatively simple)

> From the guidelines “Don’t try to design one app that combines both stickers and ridesharing, for example.”



iOS Human Interface Guidelines

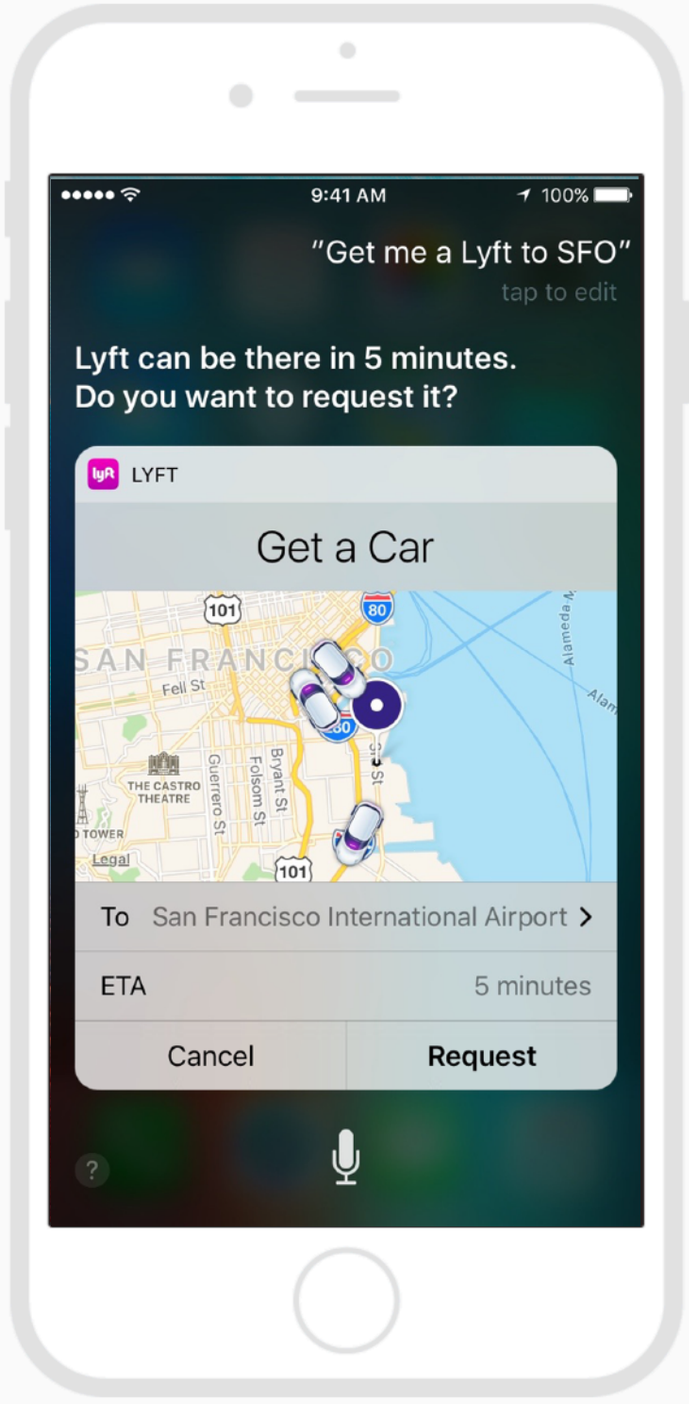
Integration with Siri

Allow users to access your app through voice controls

Can be useful for apps involving audio and video calling, messaging, payments, fitness, directions, etc.

For Siri-Enabled Apps, recommended to **minimize interaction**

Users expect a fast response (stay focused, don't provide more information than needed)



iOS Human Interface Guidelines

Search Widgets

Display notifications from you application on the user's Search an Home Screen

Very customizable (can add buttons, images, layout customization, etc.)

iOS Human Interface Guidelines

Search Widgets

Display notifications from you application on the user's Search an Home Screen

Very customizable (can add buttons, images, layout customization, etc.)

To view widgets

Search Screen > accessed by swiping to the right on Home or Lock Screen

iOS Human Interface Guidelines

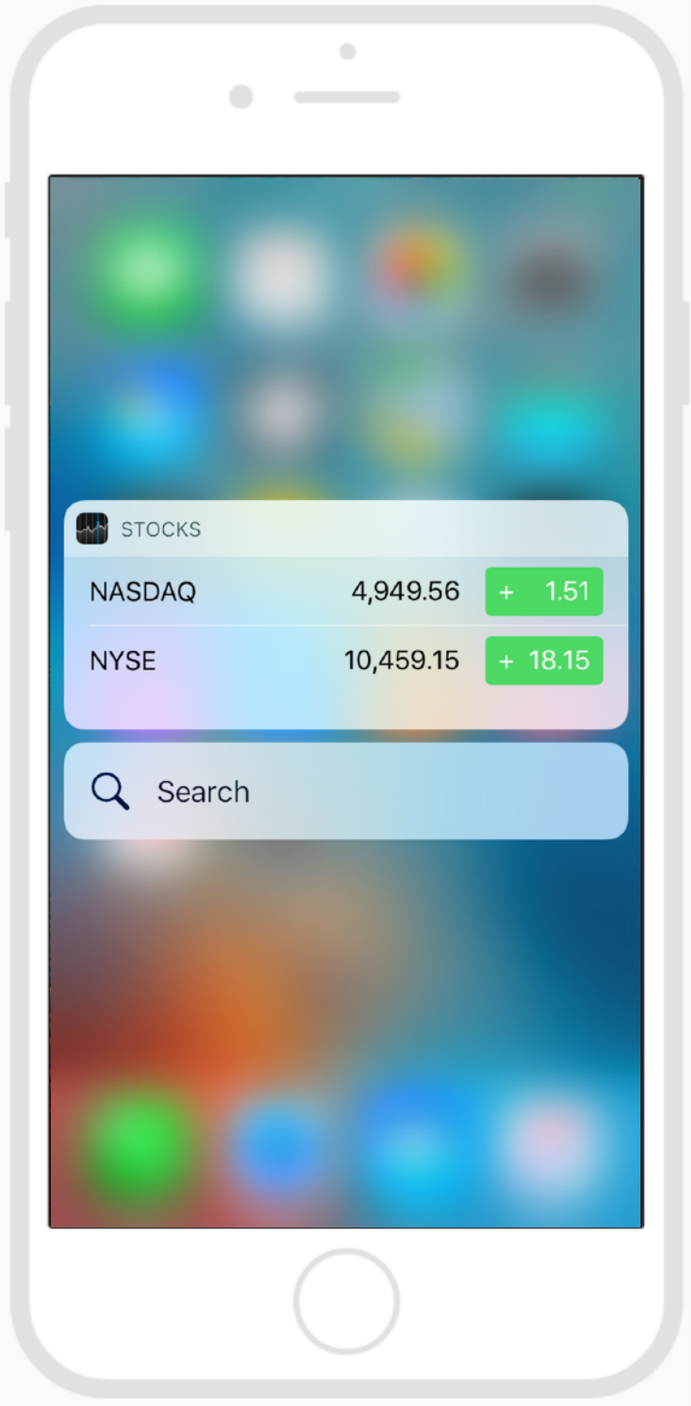
Search Widgets

Display notifications from you application on the user's Search an Home Screen

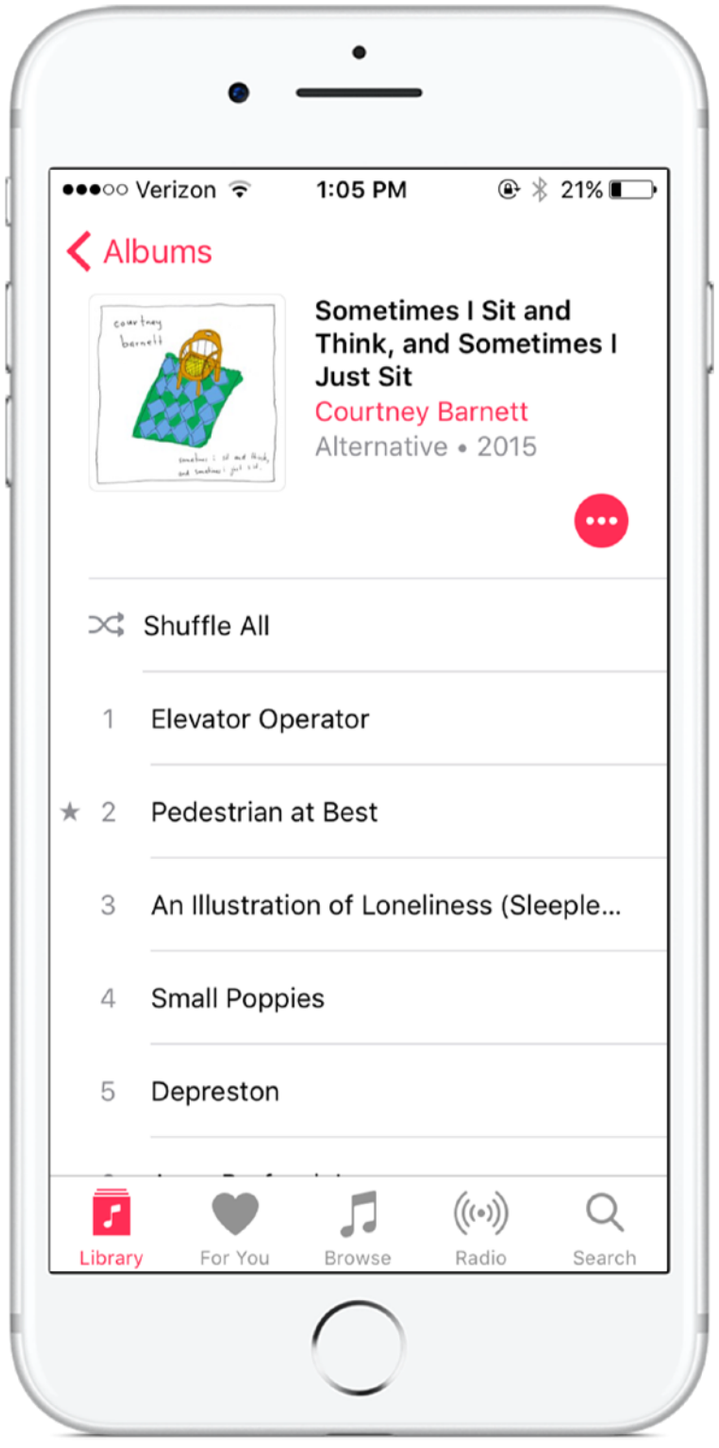
Very customizable (can add buttons, images, layout customization, etc.)

To view widgets

Home Screen > apply pressure on an app icon using 3D Touch



iOS HIG : Interface Terminology



Bars

Lets your users know “where” they are in their application. May contain buttons to trigger navigation (segues) and titles to clarify location in app

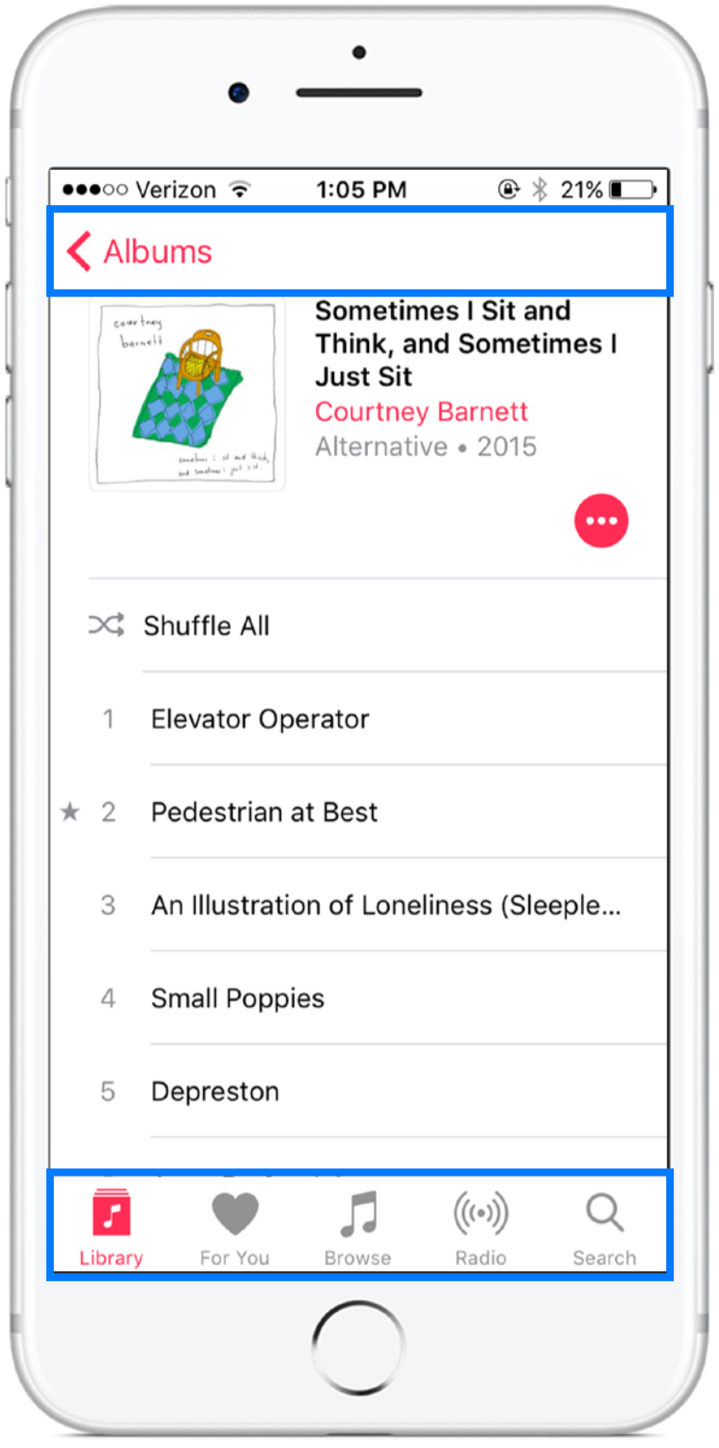
Views

Contain the content of what the users sees. This includes both the entire “screen” visible, as well as the other subviews (text, graphics, etc.)

Controls

Buttons, text fields, segmented controls, pickers,

iOS HIG : Interface Terminology



Bars

Lets your users know “where” they are in their application. May contain buttons to trigger navigation (segues) and titles to clarify location in app

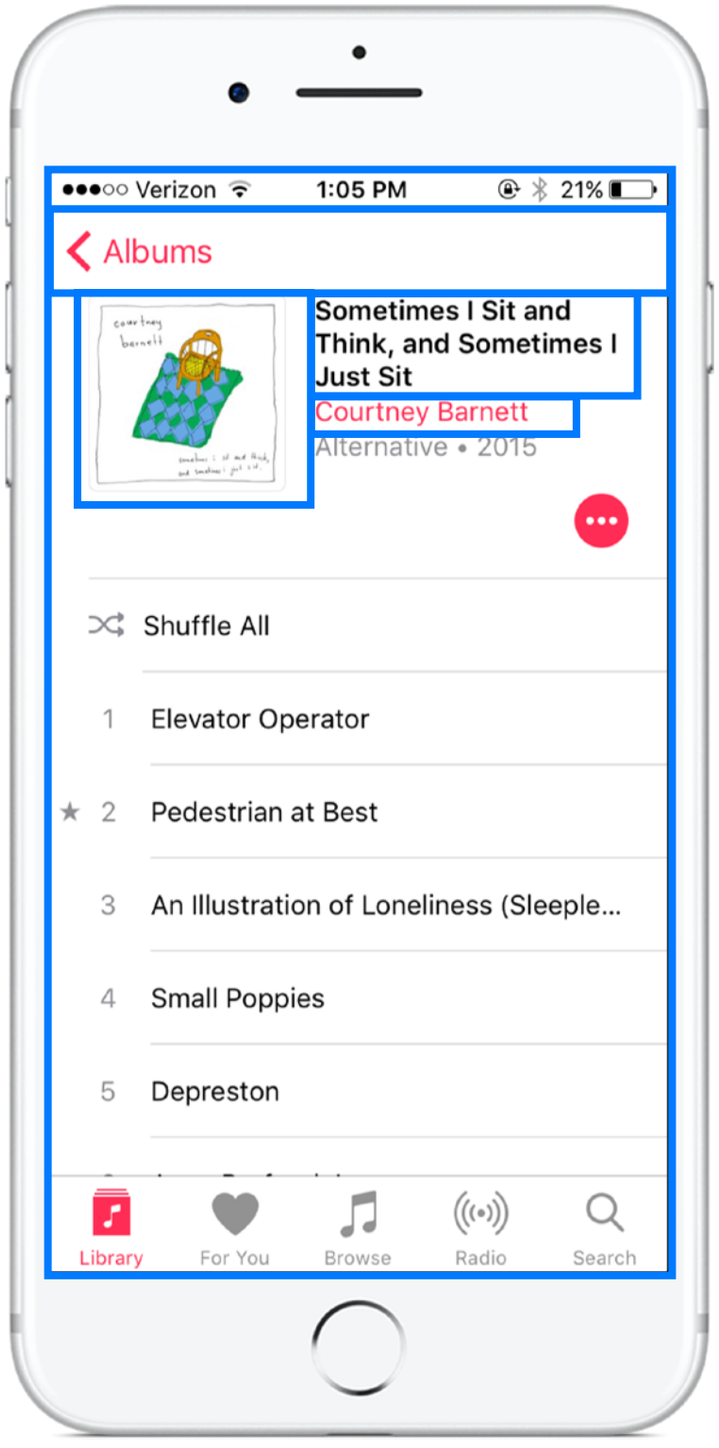
Views

Contain the content of what the users sees. This includes both the entire “screen” visible, as well as the other subviews (text, graphics, etc.)

Controls

Buttons, text fields, segmented controls, pickers,

iOS HIG : Interface Terminology



Bars

Lets your users know “where” they are in their application. May contain buttons to trigger navigation (segues) and titles to clarify location in app

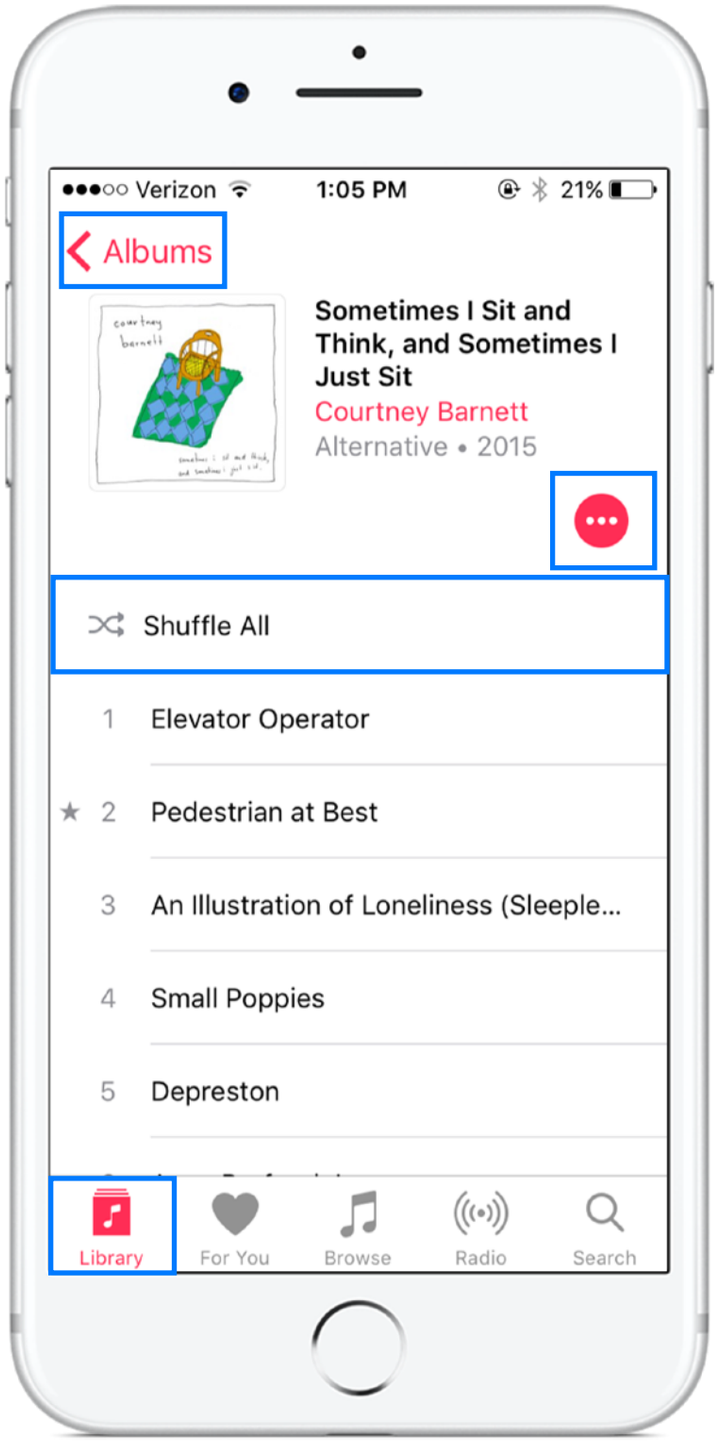
Views

Contain the content of what the users sees. This includes both the entire “screen” visible, as well as the other subviews (text, graphics, etc.)

Controls

Buttons, text fields, segmented controls, pickers,

iOS HIG : Interface Terminology



Bars

Lets your users know “where” they are in their application. May contain buttons to trigger navigation (segues) and titles to clarify location in app

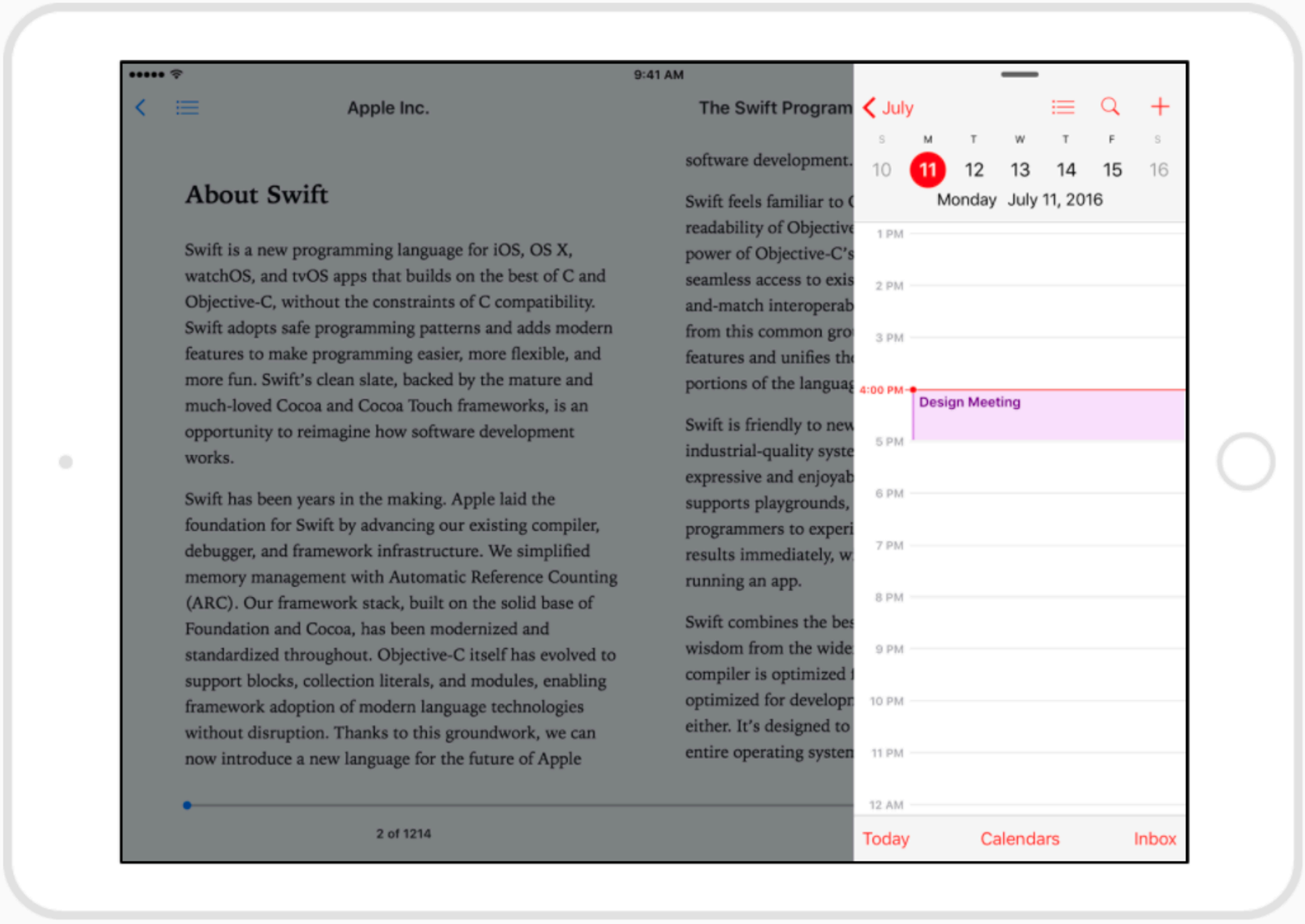
Views

Contain the content of what the users sees. This includes both the entire “screen” visible, as well as the other subviews (text, graphics, etc.)

Controls

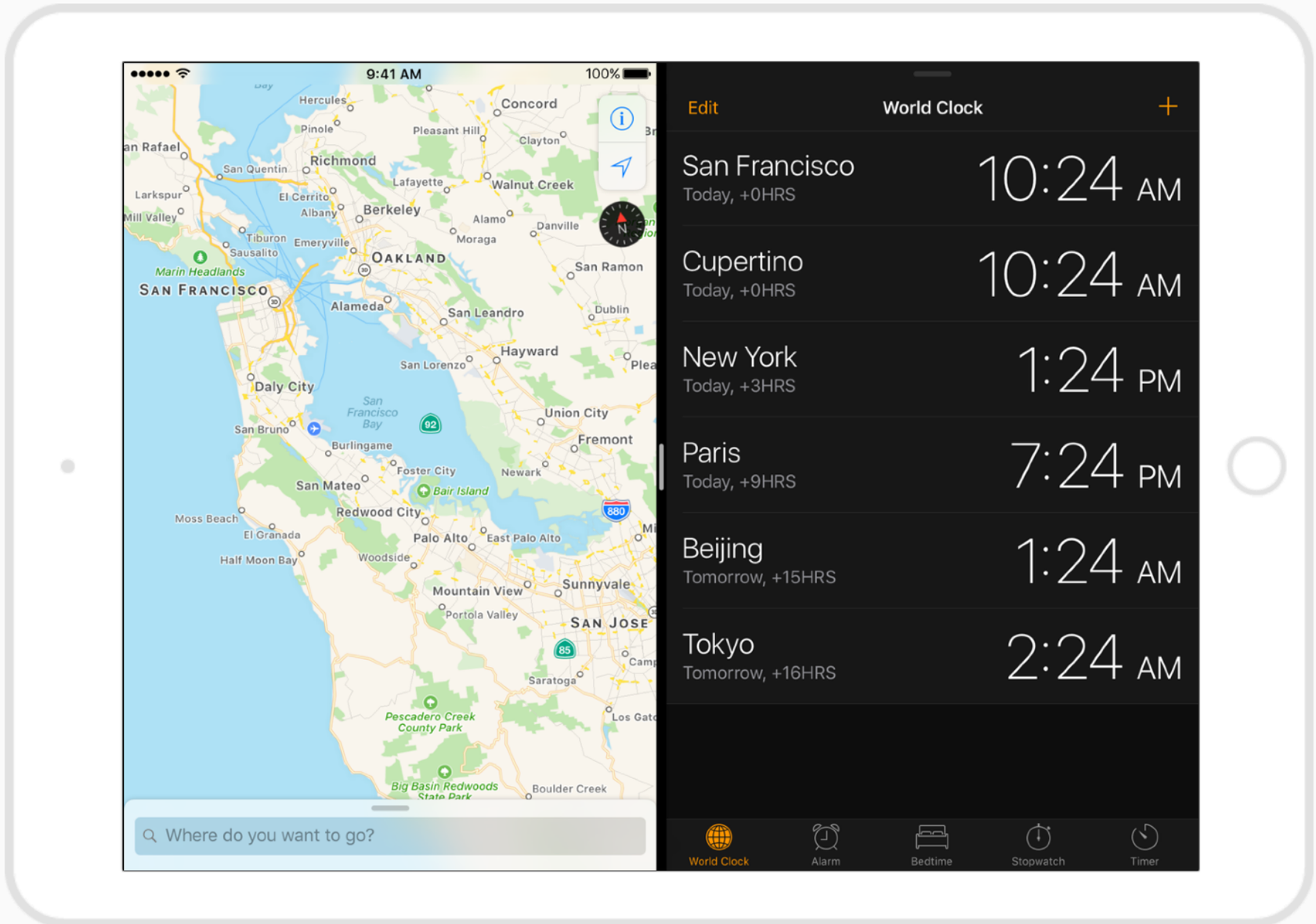
Buttons, text fields, segmented controls, etc.

iOS HIG : Multitasking (iPad)



Designing with Multitasking in mind (example Slide Over)

iOS HIG : Multitasking (iPad)



Designing with Multitasking in mind (example Slide View)

iOS HIG : Branding

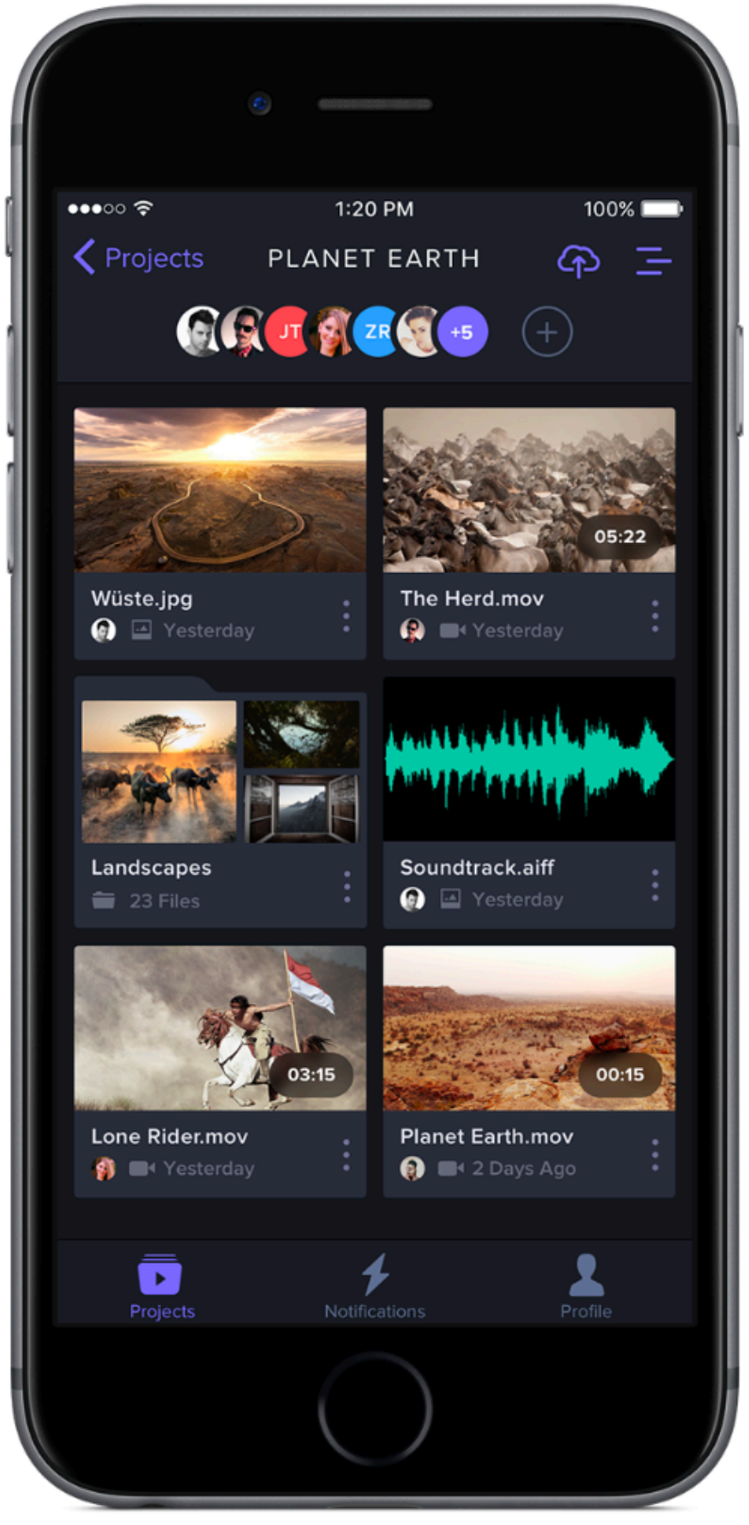
While it is important to have an app “image” or “brand”, avoid over-using logos, icon images, etc.

Examples:

No need to include logo in every view of your application

Focus on design schemes (fonts, colors, layouts) rather than overt branding

Avoid sacrificing screen space for your brand unless necessary



Frame.io
Video Collaboration

iOS HIG : Branding

While it is important to have an app “image” or “brand”, avoid over-using logos, icon images, etc.

Examples:

No need to include logo in every view of your application

Focus on design schemes (fonts, colors, layouts) rather than overt branding

Avoid sacrificing screen space for your brand unless necessary

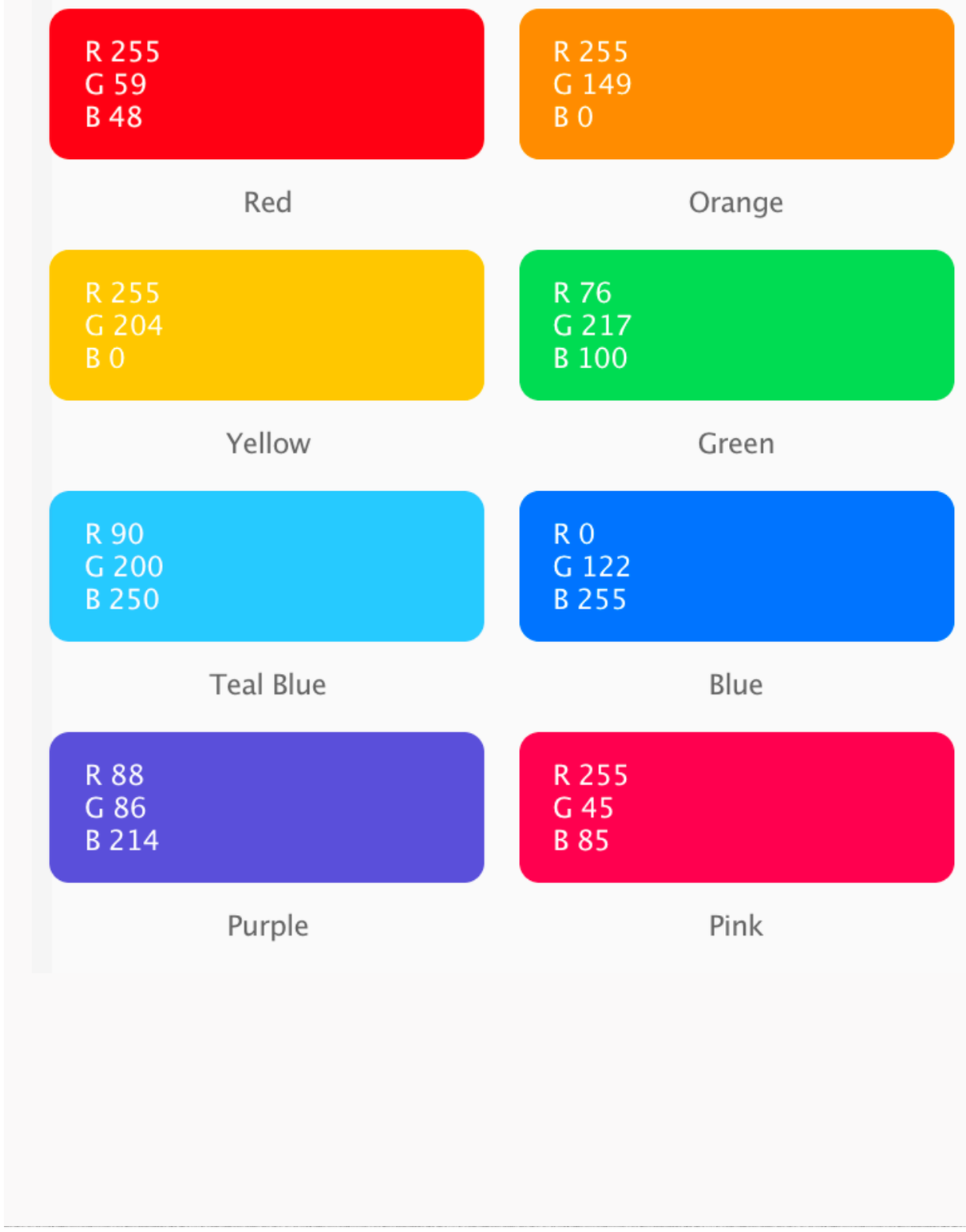
INKS
State of Play Games

iOS HIG : Color

Add cohesion to your app by defining a consistent color scheme

Distinguish between interactive and un-interactive UI elements using color

Create color constants to be used throughout your application by creating `UIColor` objects

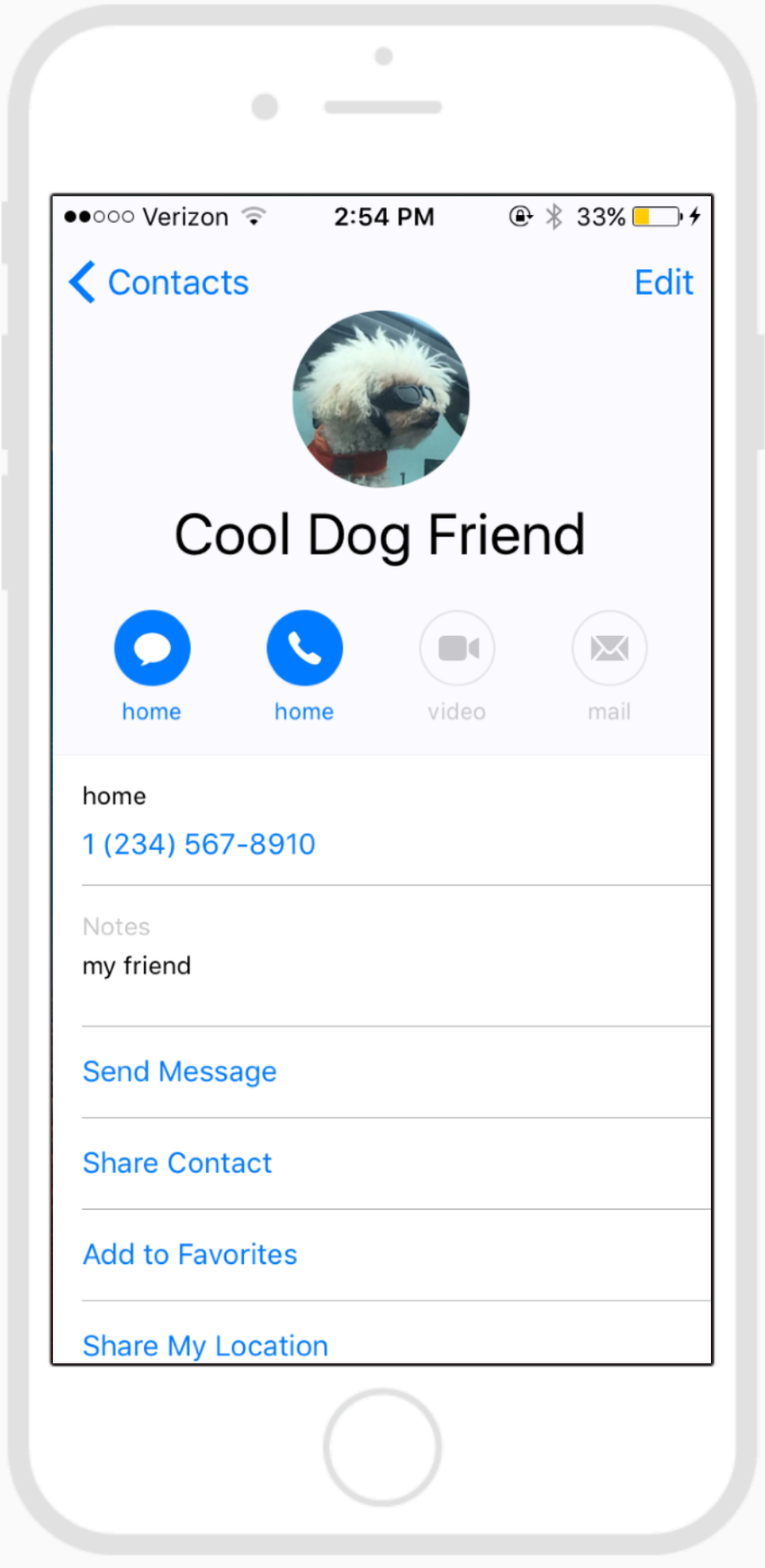


iOS HIG : Color

Add cohesion to your app by defining a consistent color scheme

Distinguish between interactive and un-interactive UI elements using color

Create color constants to be used throughout your application by creating `UIColor` objects

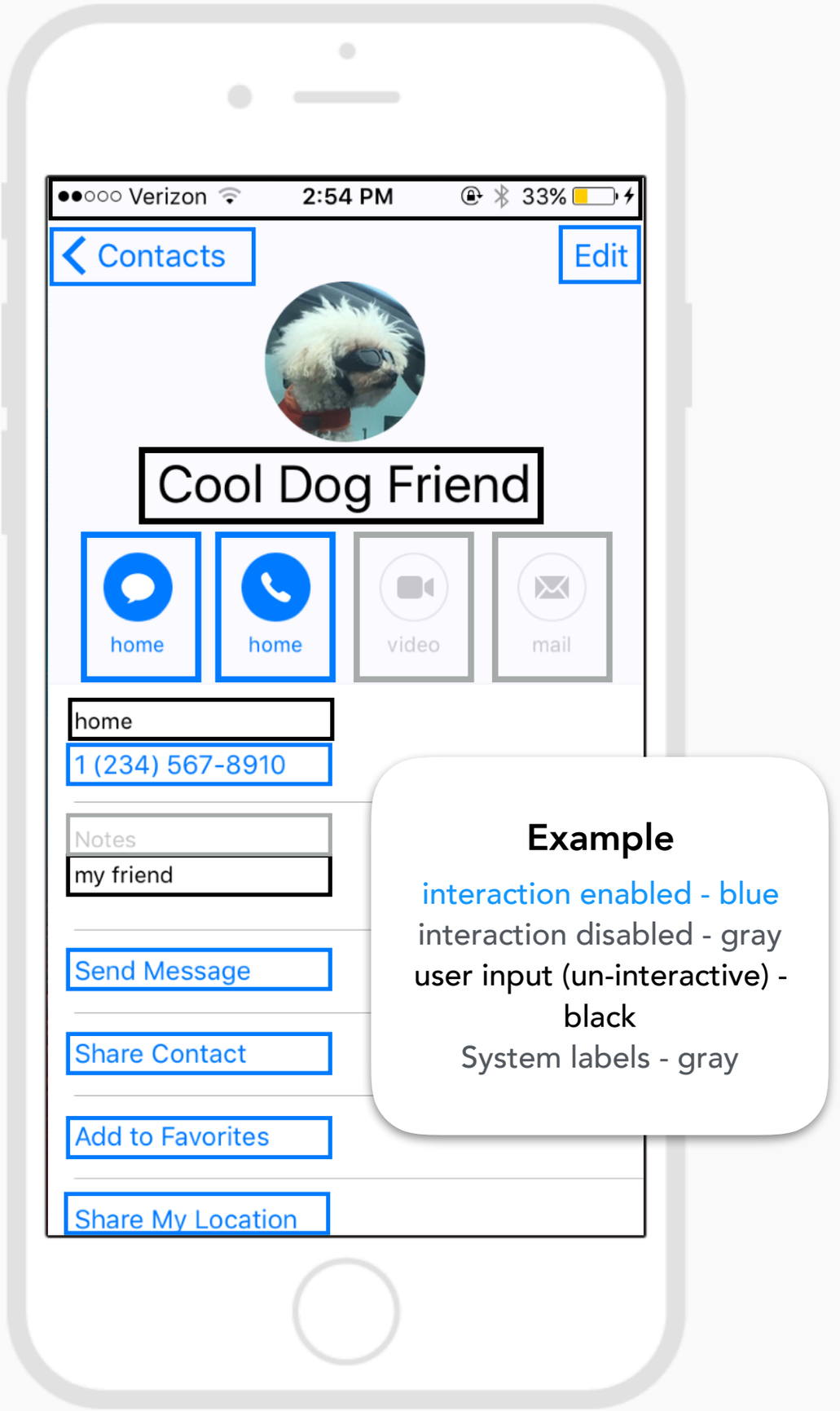


iOS HIG : Color

Add cohesion to your app by defining a consistent color scheme

Distinguish between interactive and un-interactive UI elements using color

Create color constants to be used throughout your application by creating `UIColor` objects



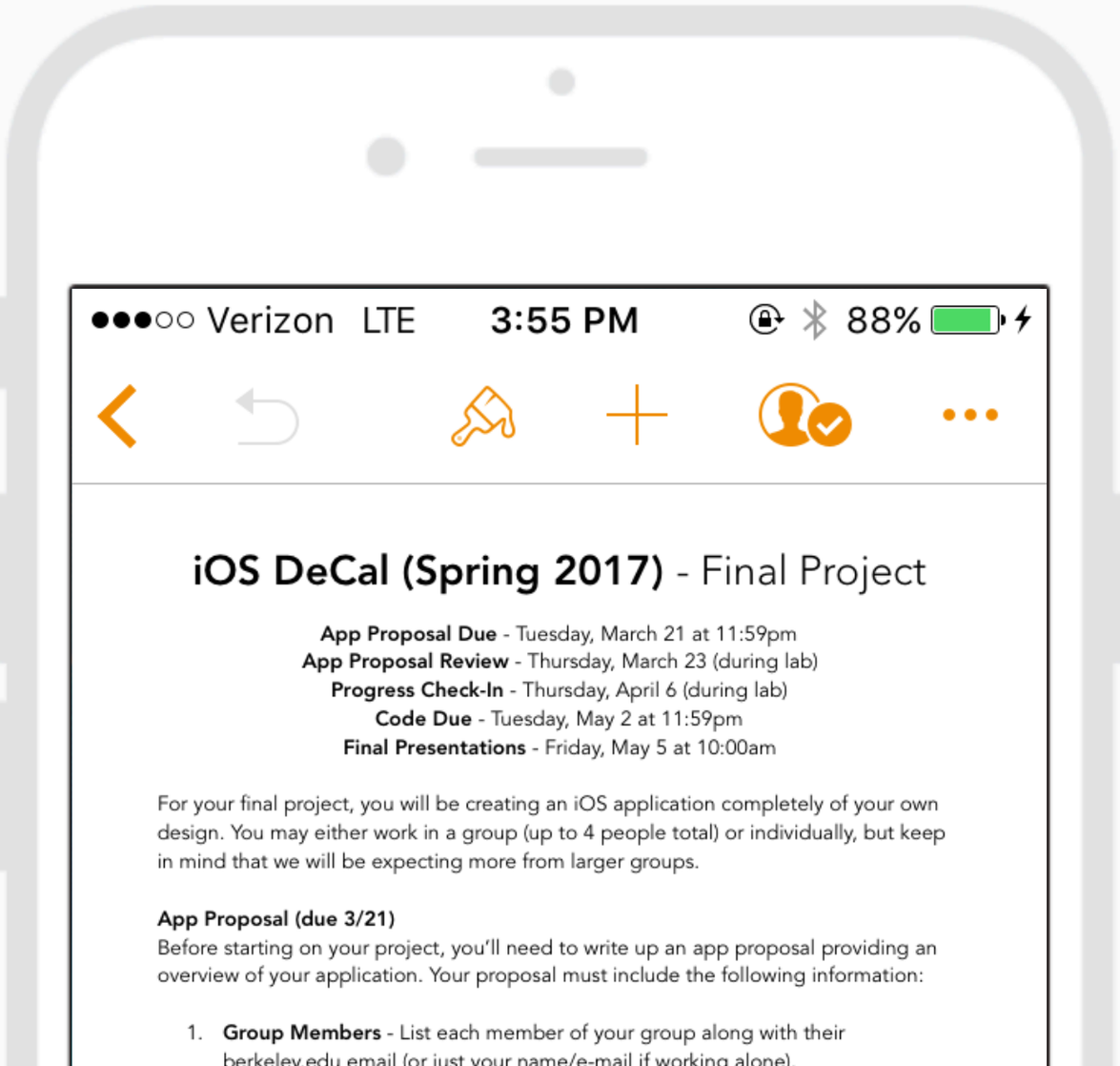
iOS HIG : Color

Add cohesion to your app by defining a consistent color scheme

Distinguish between interactive and un-interactive UI elements using color

Create color constants to be used throughout your application by creating `UIColor` objects

Common iOS Design practice to set “Enabled Color” as your app’s brand color



iOS HIG : Color

Add cohesion to your app by defining a consistent color scheme

Distinguish between interactive and un-interactive UI elements using color

Create color constants to be used throughout your application by creating `UIColor` objects

Creating a `UIColor` object with Predefined Colors

```
class var black: UIColor
```

A color object in the sRGB color space whose grayscale value is 0.0 and whose alpha value is 1.0.

```
class var blue: UIColor
```

A color object whose RGB values are 0.0, 0.0, and 1.0 and whose alpha value is 1.0.

```
class var brown: UIColor
```

A color object whose RGB values are 0.6, 0.4, and 0.2 and whose alpha value is 1.0.

```
class var clear: UIColor
```

A color object whose grayscale and alpha values are both 0.0.

```
class var cyan: UIColor
```

A color object whose RGB values are 0.0, 1.0, and 1.0 and whose alpha value is 1.0.

```
class var darkGray: UIColor
```

A color object whose grayscale value is 1/3 and whose alpha value is 1.0.

… and more (see UIColor)

iOS HIG : Color

Add cohesion to your app by defining a consistent color scheme

Distinguish between interactive and un-interactive UI elements using color

Create color constants to be used throughout your application by creating `UIColor` objects

Creating a Custom `UIColor` object using Color Spaces

```
init(white: CGFloat, alpha: CGFloat)
```

Initializes and returns a color object using the specified opacity and grayscale values.

```
init(hue: CGFloat, saturation: CGFloat, brightness: CGFloat, alpha: CGFloat)
```

Initializes and returns a color object using the specified opacity and HSB color space component values.

```
init(red: CGFloat, green: CGFloat, blue: CGFloat, alpha: CGFloat)
```

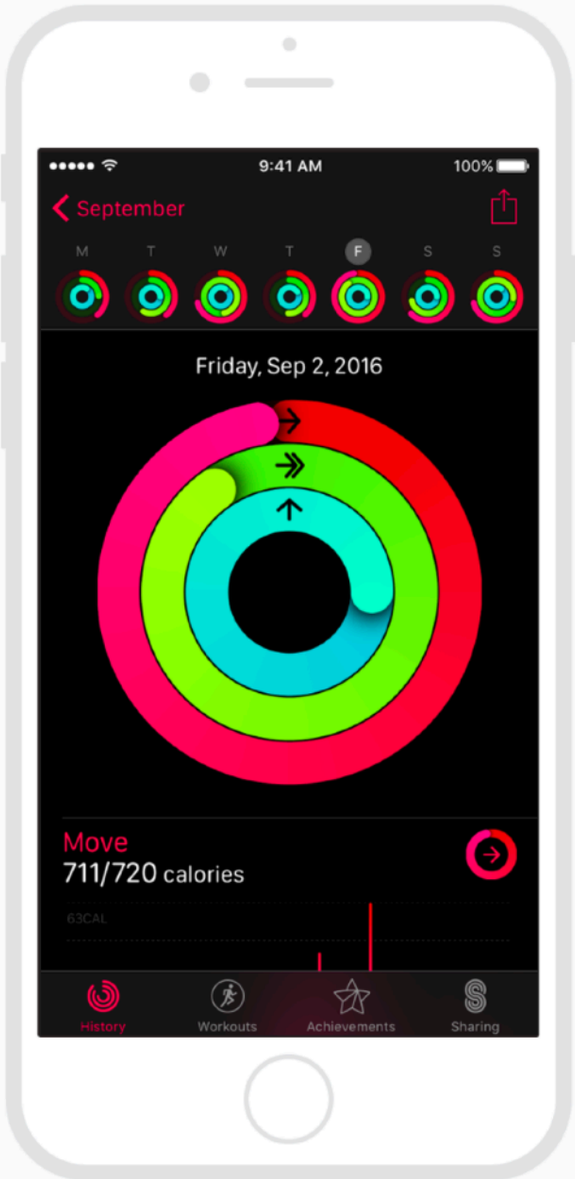
Initializes and returns a color object using the specified opacity and RGB component values.

```
init(displayP3Red: CGFloat, green: CGFloat, blue: CGFloat, alpha: CGFloat)
```

Initializes and returns a color object using the specified opacity and RGB component values in the Display P3 color space.

iOS HIG : Color

App Colors
(standard)

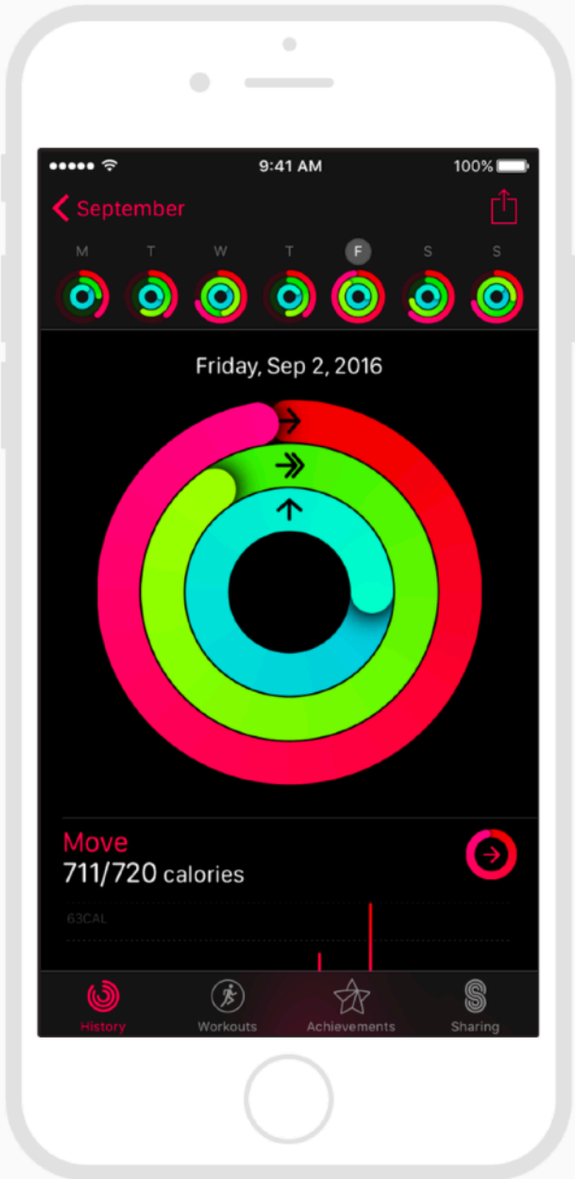


App Colors
(without red-green)

Keep in mind what your app will look like for users with various types of color vision impairment

iOS HIG : Color

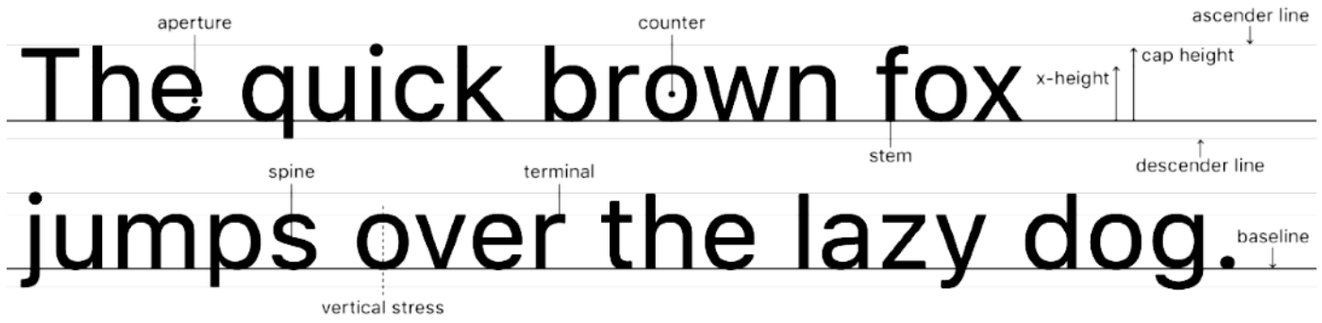
App Colors
(standard)



App Colors
(without red-green)

Photoshop has accessibility color filters to help you do this
http://www.adobe.com/accessibility/products/photoshop.html

iOS HIG : Fonts and Typography



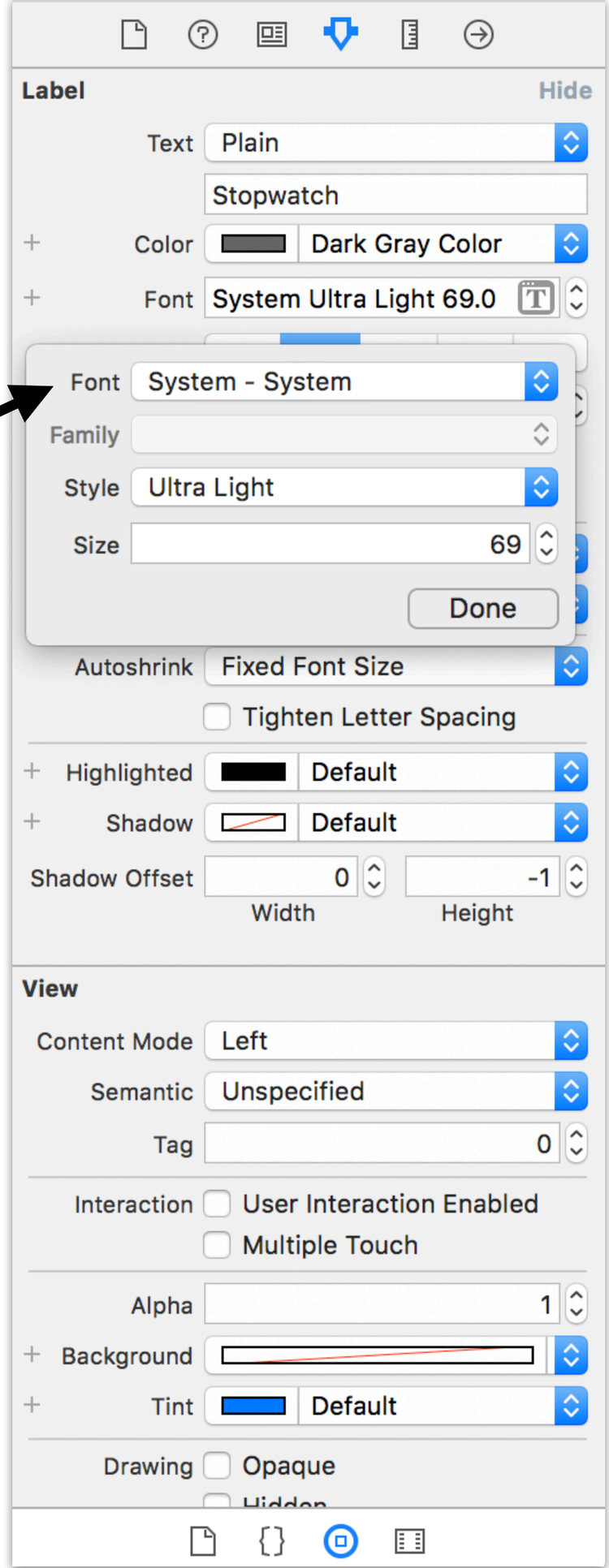
San Francisco

The System Font for iOS

Created by a team at Apple in 2014

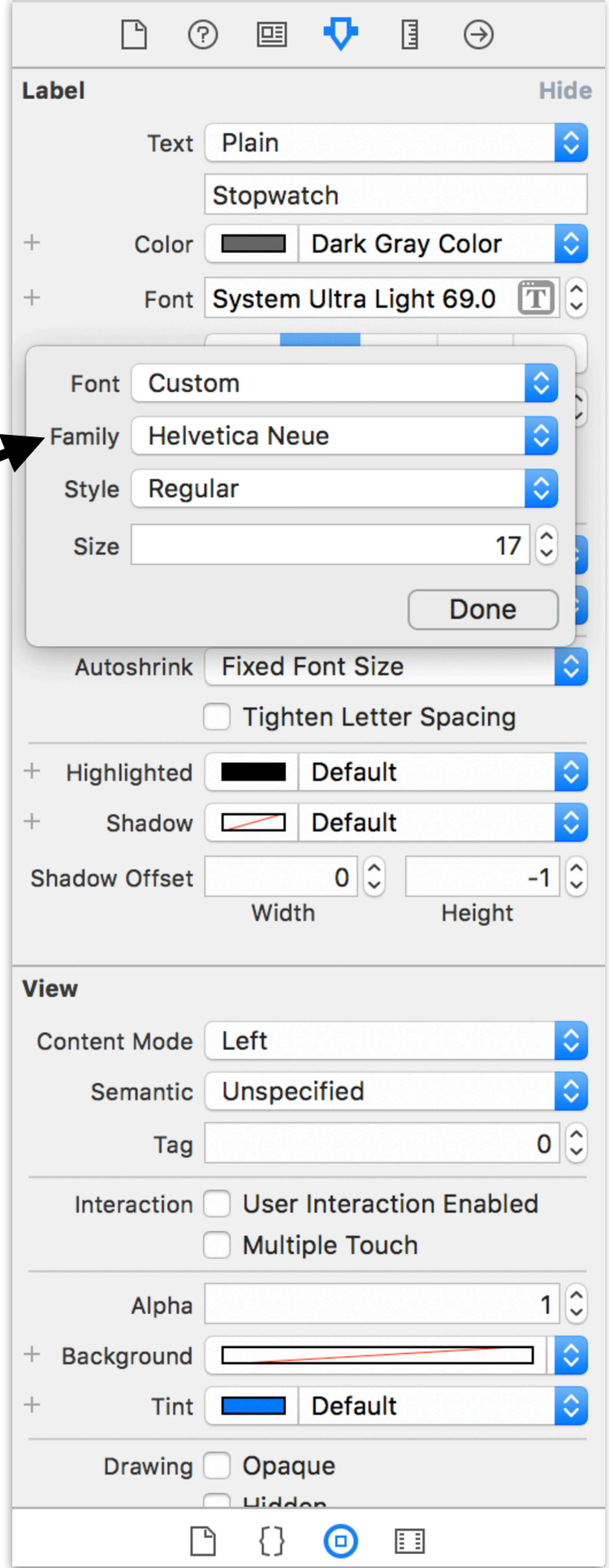
iOS HIG : Fonts

When you add new UI elements with text to your app, the font family will default to **System** (San Francisco)



iOS HIG : Fonts

Set Font to "**Custom**" to change to a different Font Family



iOS HIG : Fonts

Generally, try to stick to one font throughout your entire app

- Instead of using different fonts, try experimenting with a few different font styles, weights, and sizes (all within the same font family)

Example: Helvetica Neue typeface weights

Helvetica Neue Thin

Helvetica Neue Light

Helvetica Neue Regular

Helvetica Neue Medium

Helvetica Neue Bold

Programmatic Design

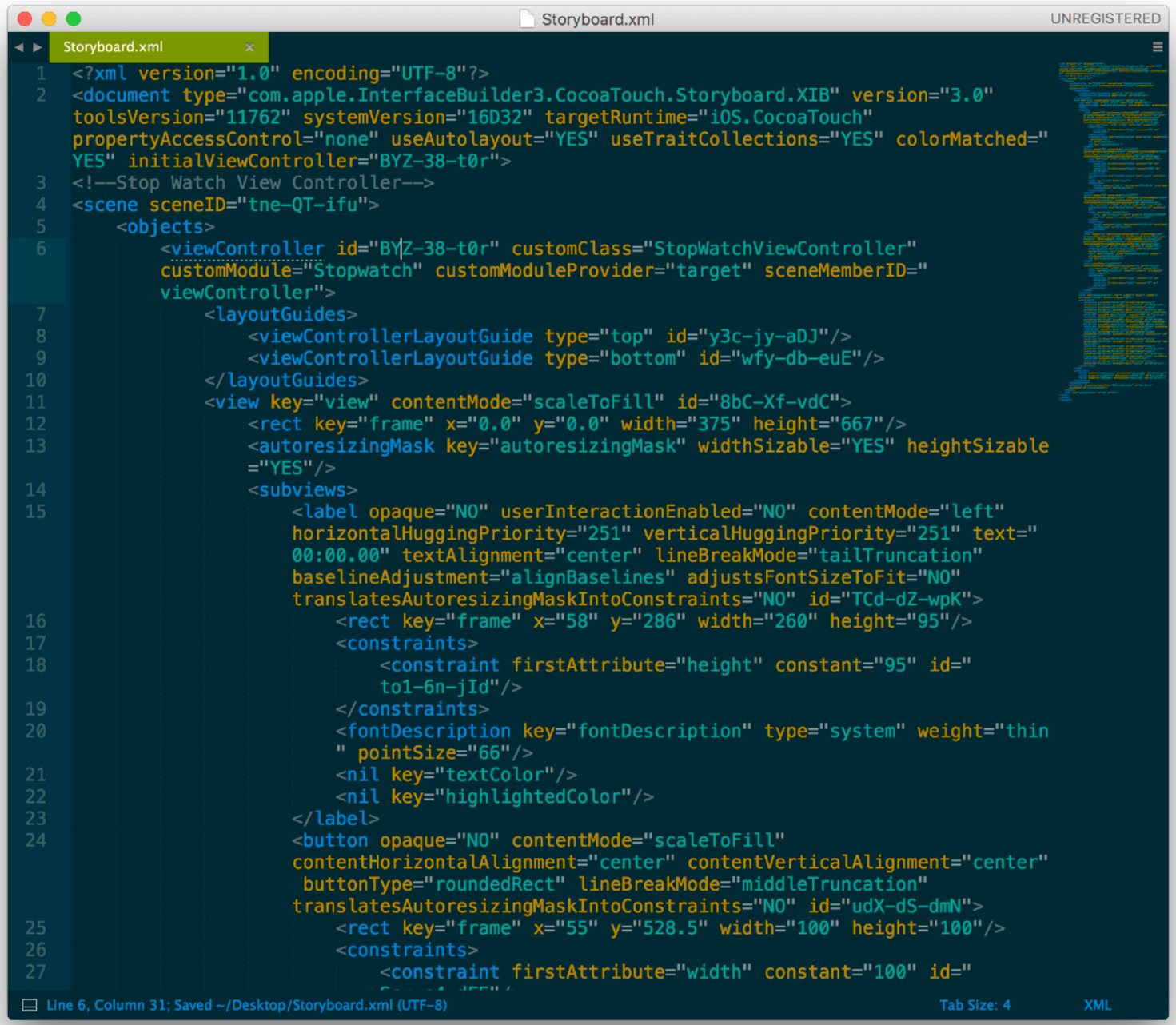
Storyboard : Review

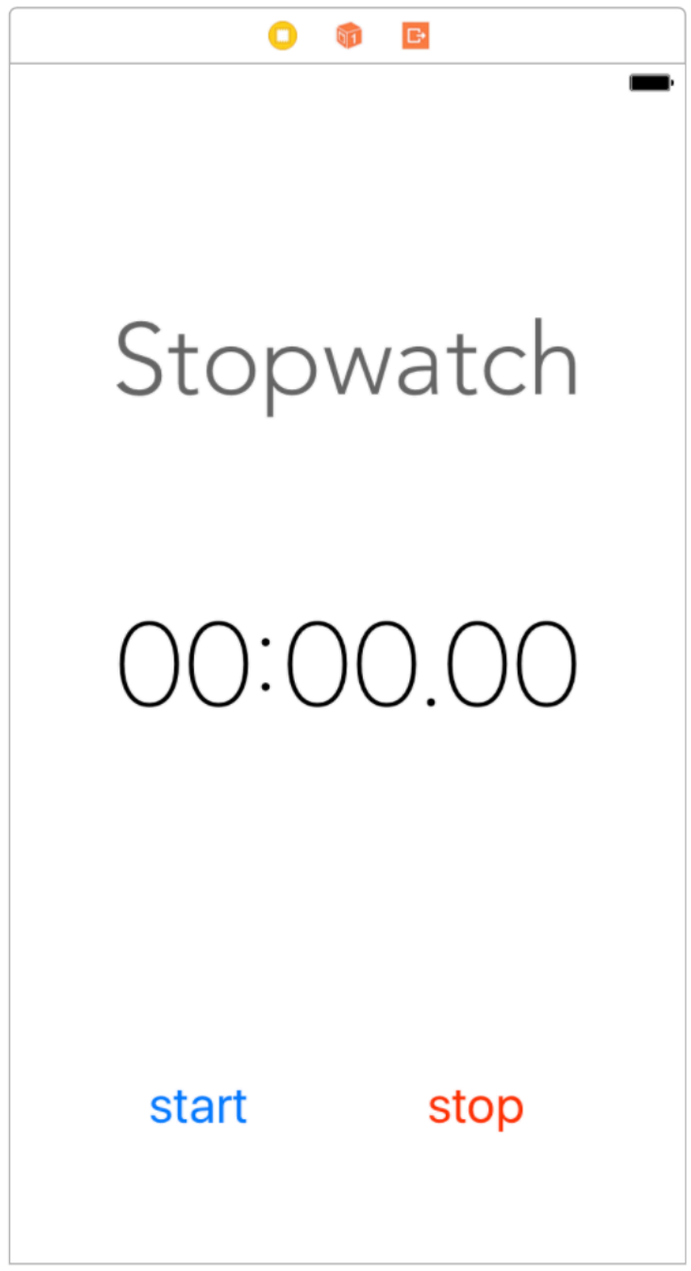
Up to now, you have been creating applications using Storyboard / Interface Builder

Pros of Storyboard

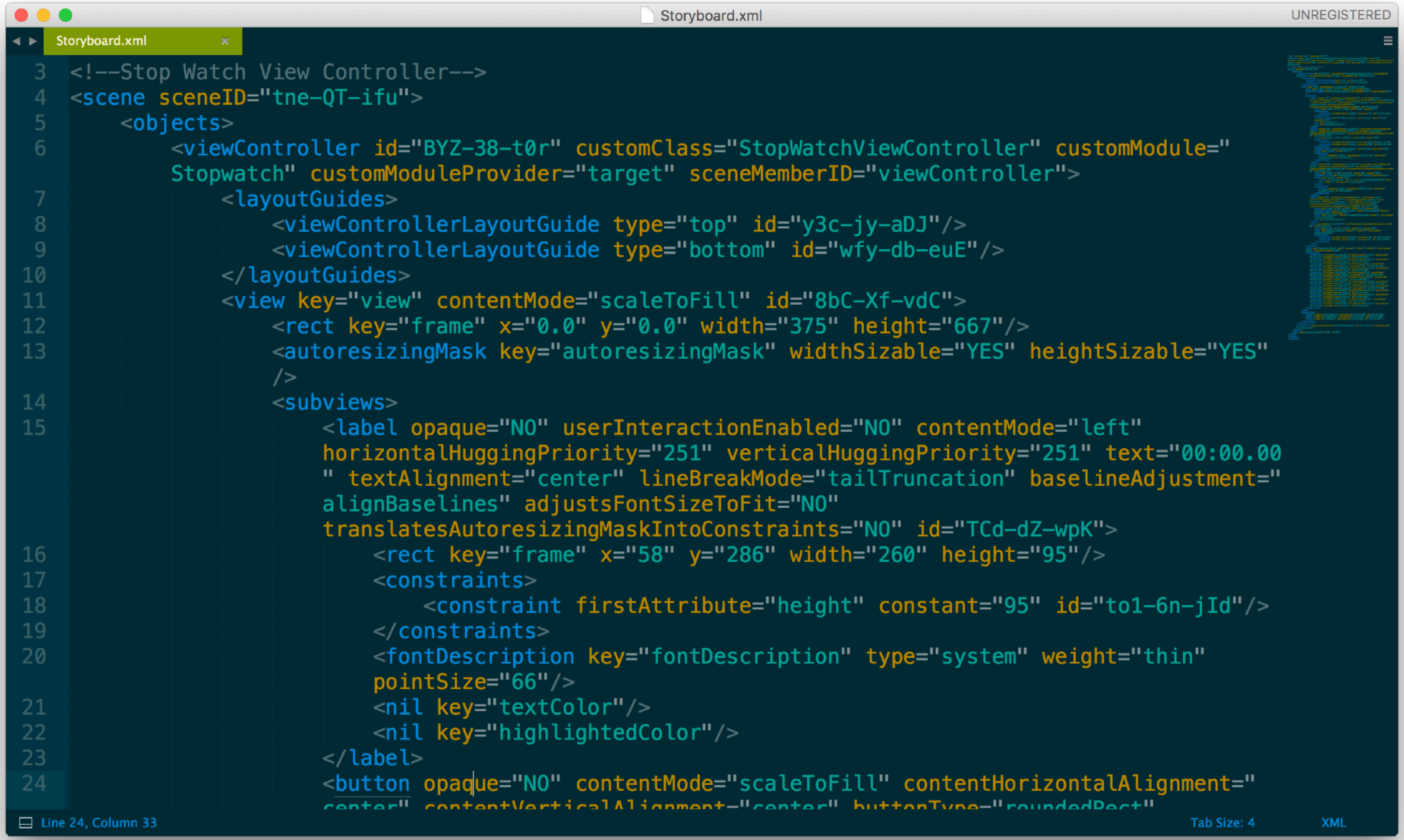
Drag and drop interface makes it really easy to visualize your application immediately
Relatively low learning curve
Great for small projects
The future of User Interface programming?

Storyboard : Beneath the hood

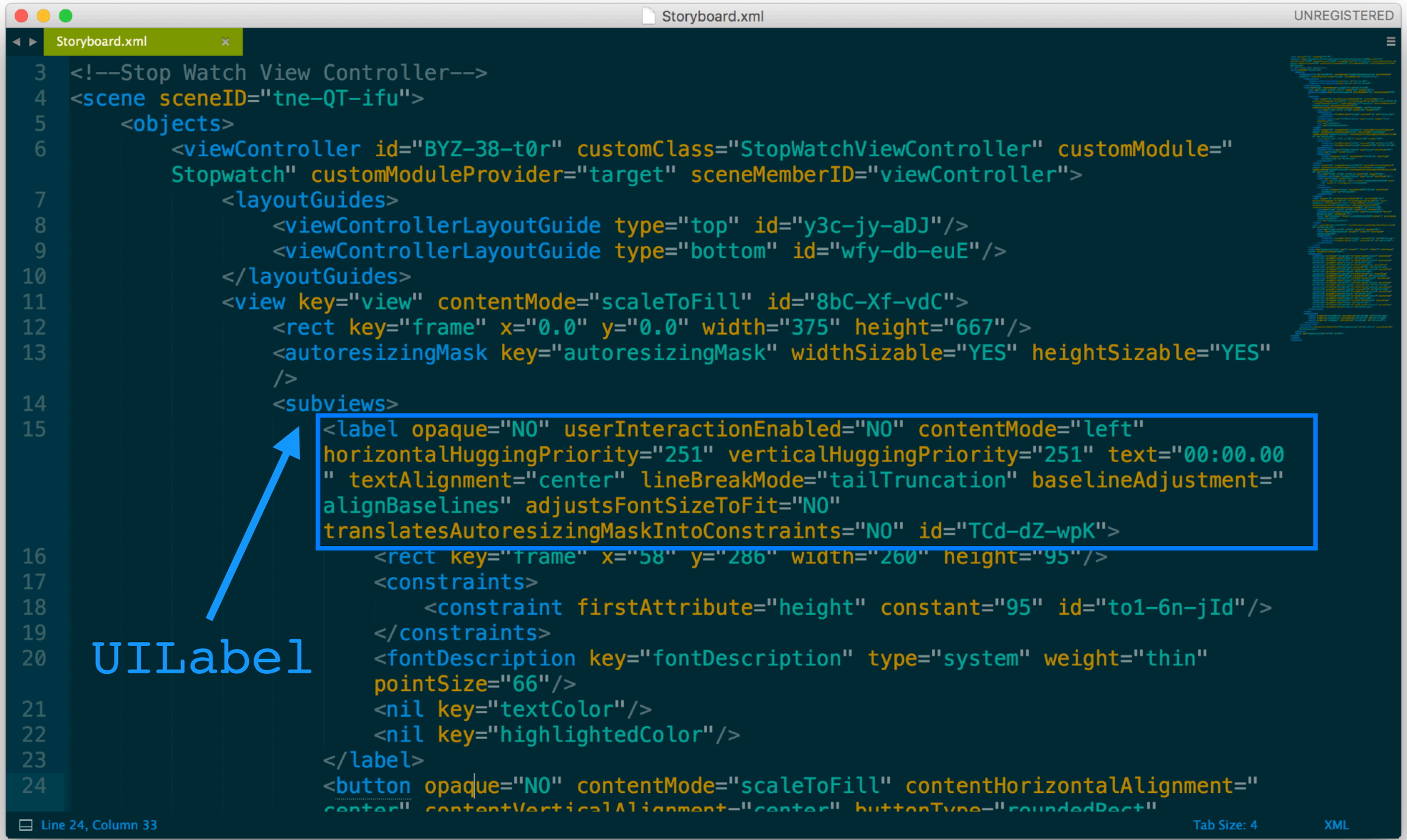




Main.storyboard files are just XML files



You can view the file generated by Interface Builder by opening up Main.storyboard in any text editor



Each time you add a button / label / constraint / etc.,
you'll be able to see it added to this file

Storyboard : Problems

Cons of using Storyboard

Easy to get cluttered for larger scale applications

XML files are prone to merge conflicts when using version control (git)

No way to define layout constants or easy way to reuse UI layouts

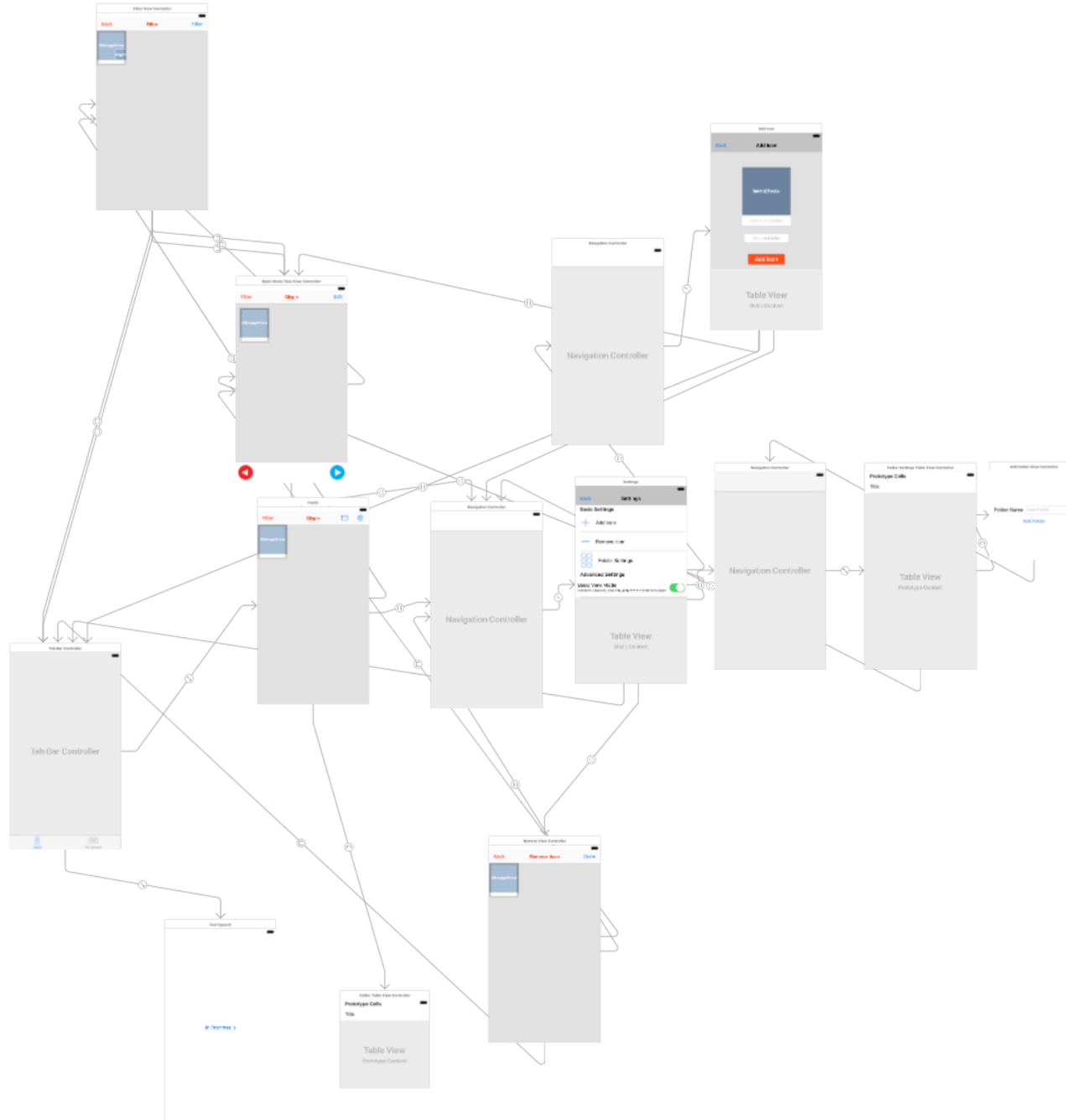
Storyboard : Problems

Cons of using Storyboard

Easy to get cluttered for larger scale applications

XML files are prone to merge conflicts when using version control (git)

No way to define layout constants or easy way to reuse UI layouts



Storyboard : Problems

Cons of using Storyboard

Easy to get cluttered for larger scale applications

XML files are prone to merge conflicts when using version control (git)

No way to define layout constants or easy way to reuse UI layouts

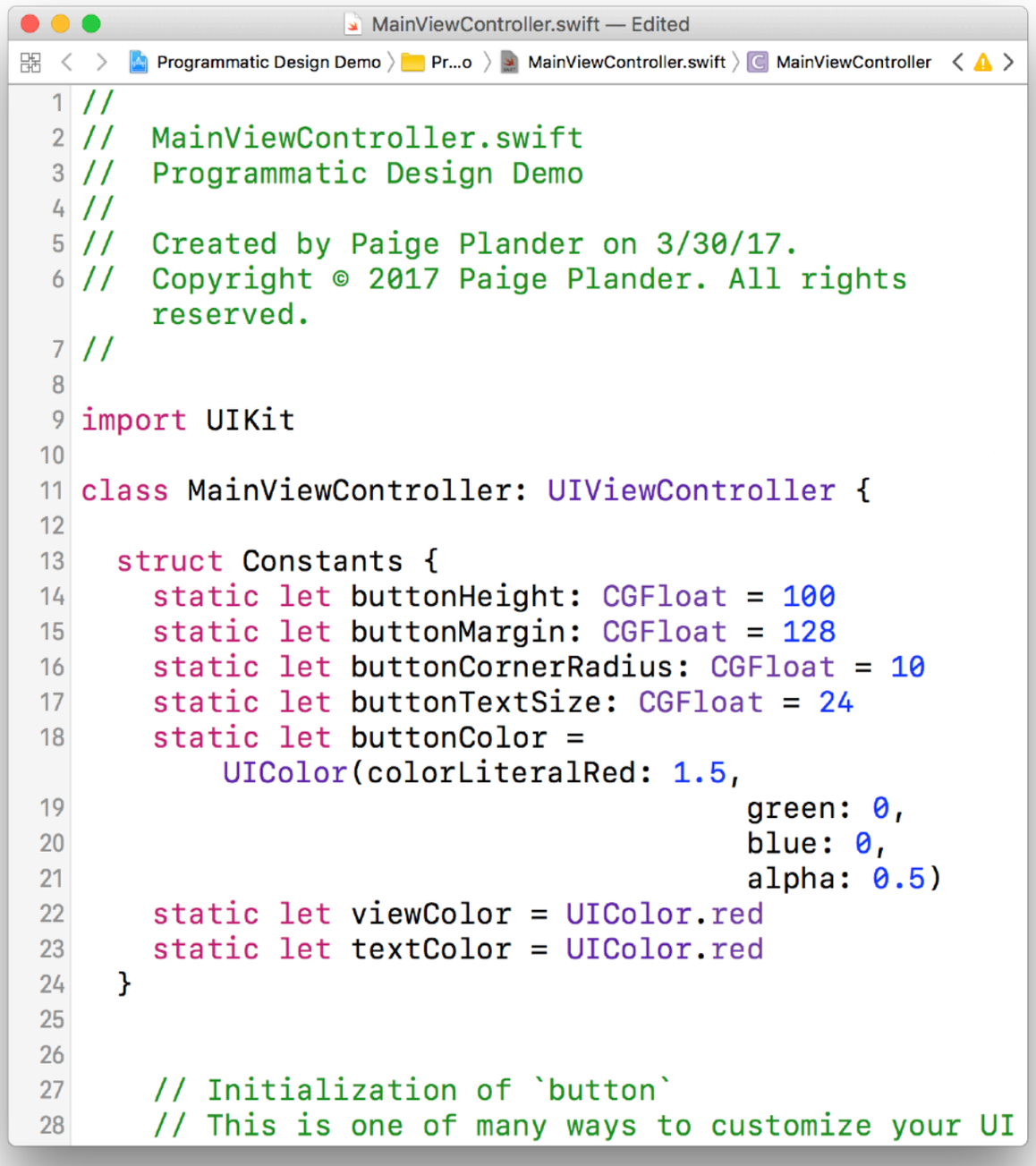
Storyboard : Problems

Cons of using Storyboard

Easy to get cluttered for larger scale applications

XML files are prone to merge conflicts when using version control (git)

No way to define layout constants or easy way to reuse UI layouts

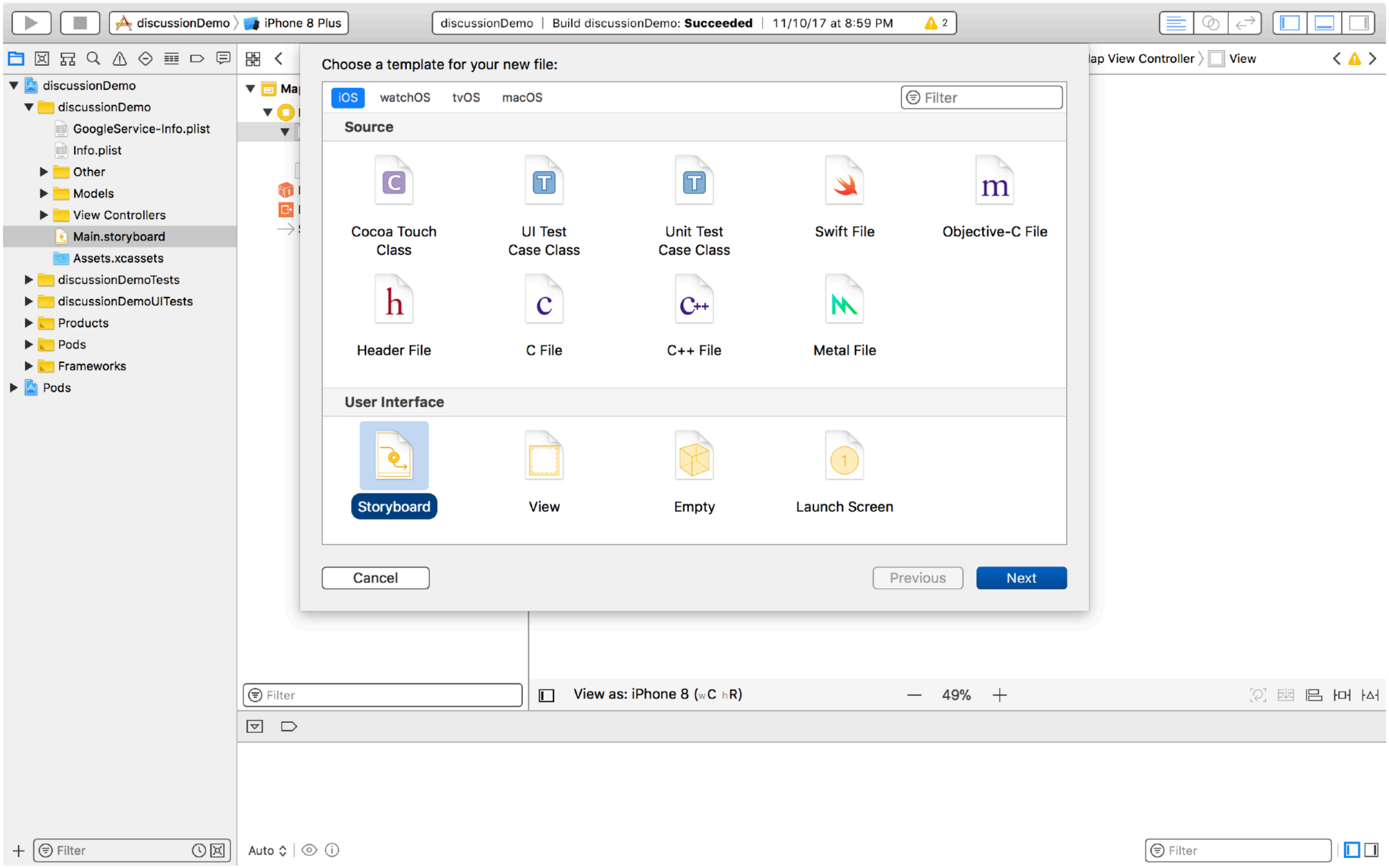


Storyboards References

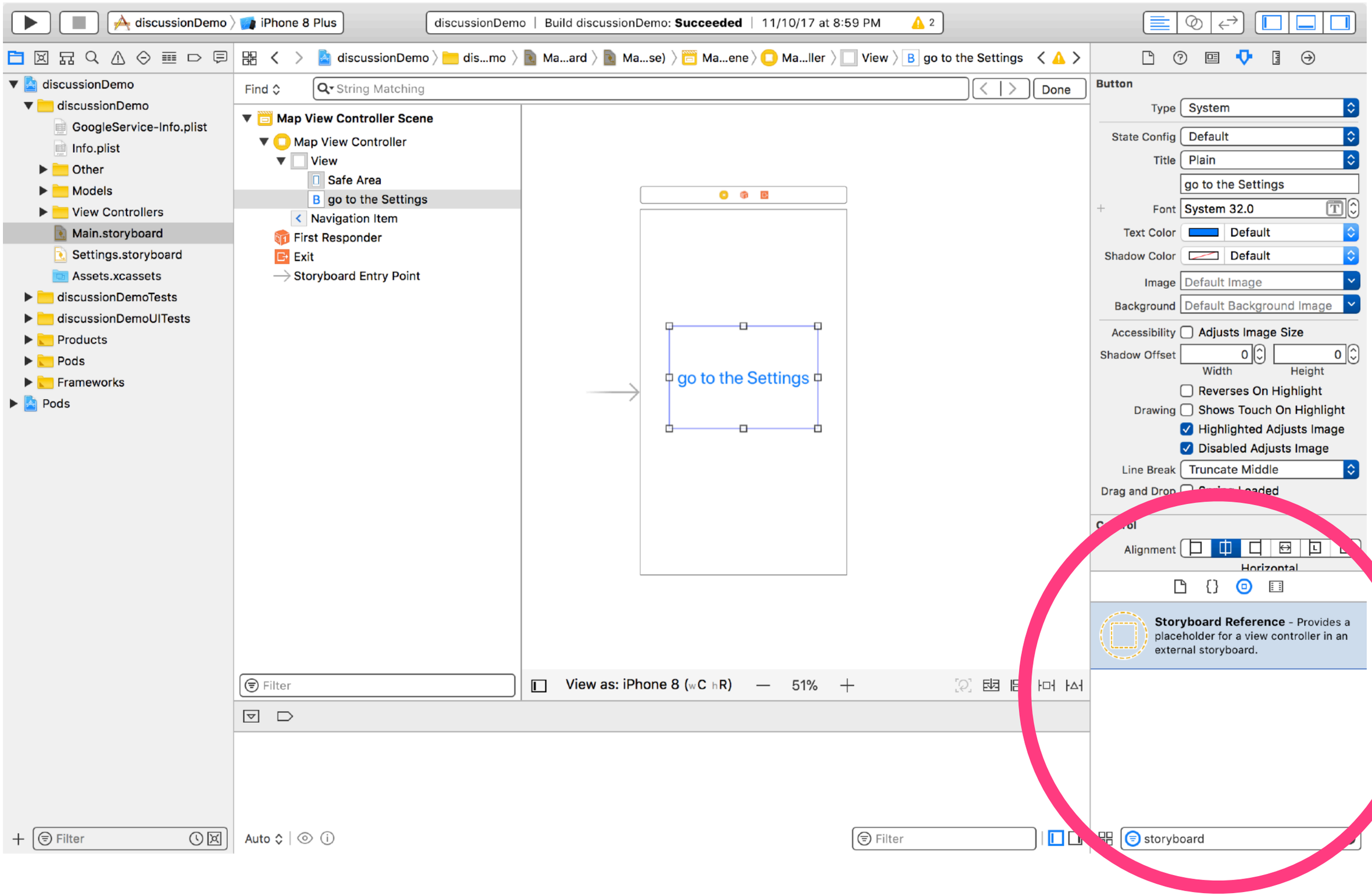
If you have a large project, break it up into several different Storyboard files.

Use “Storyboard References” to transition between Storyboards

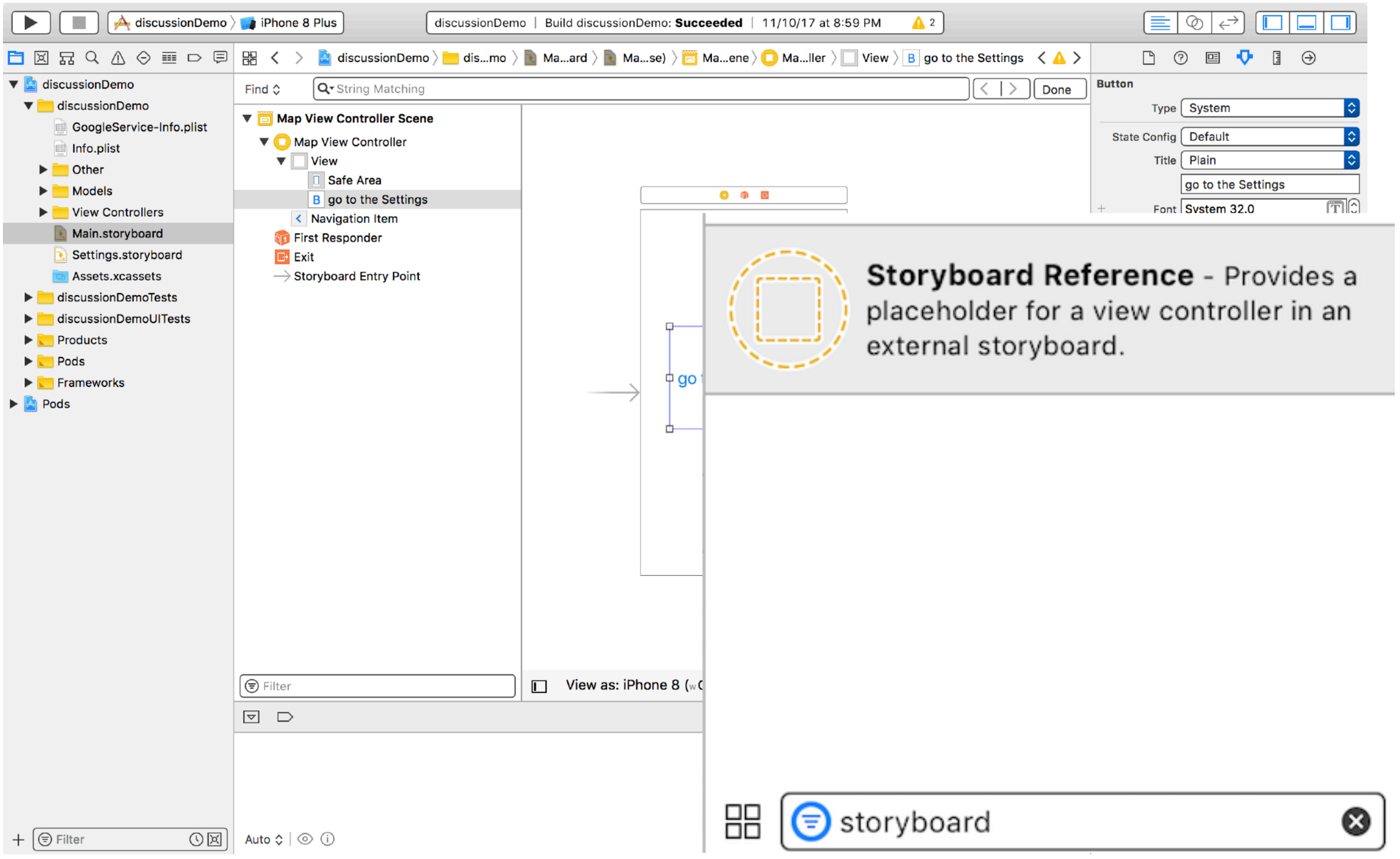
Create the storyboard file



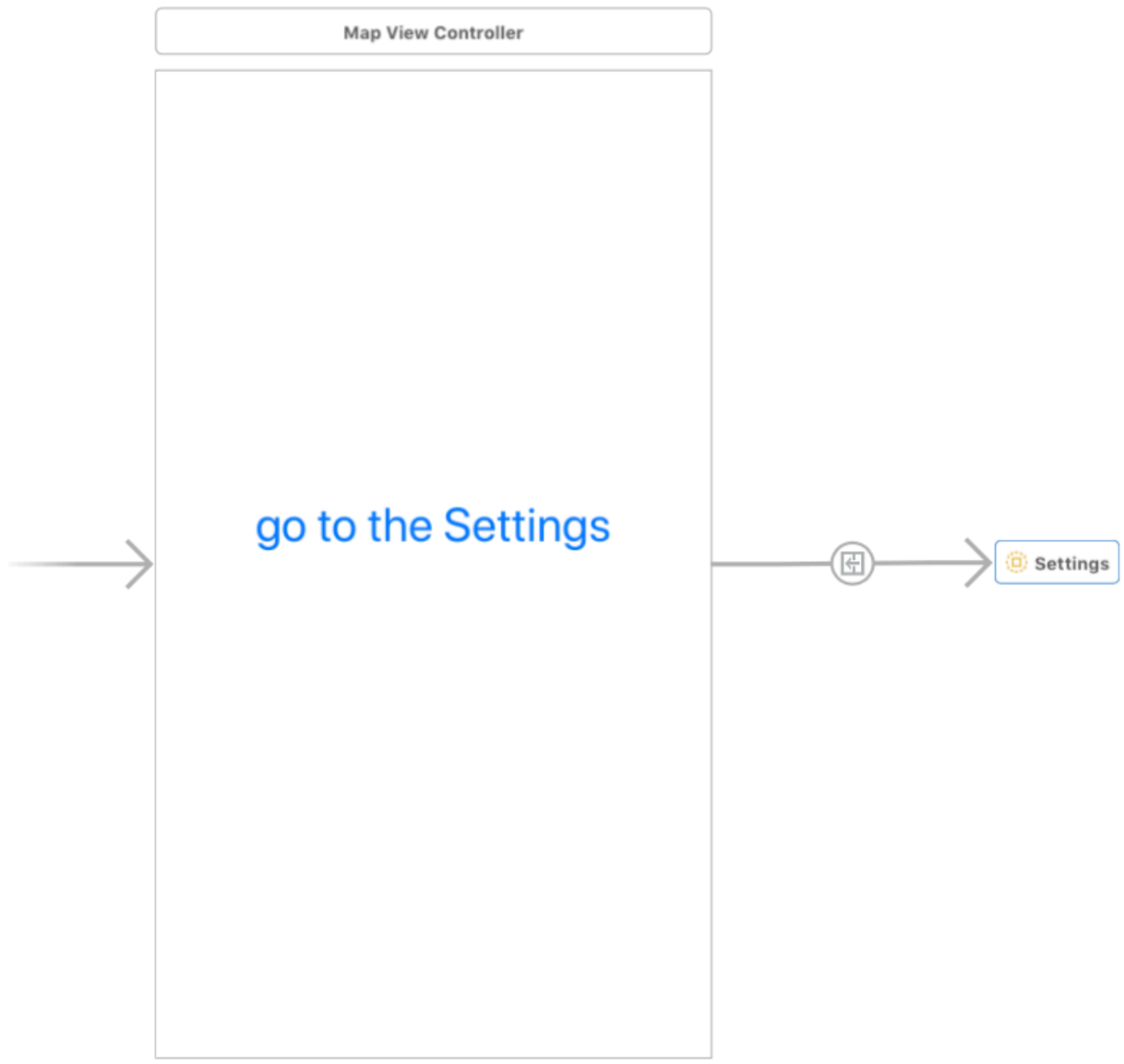
Create a Storyboard Reference

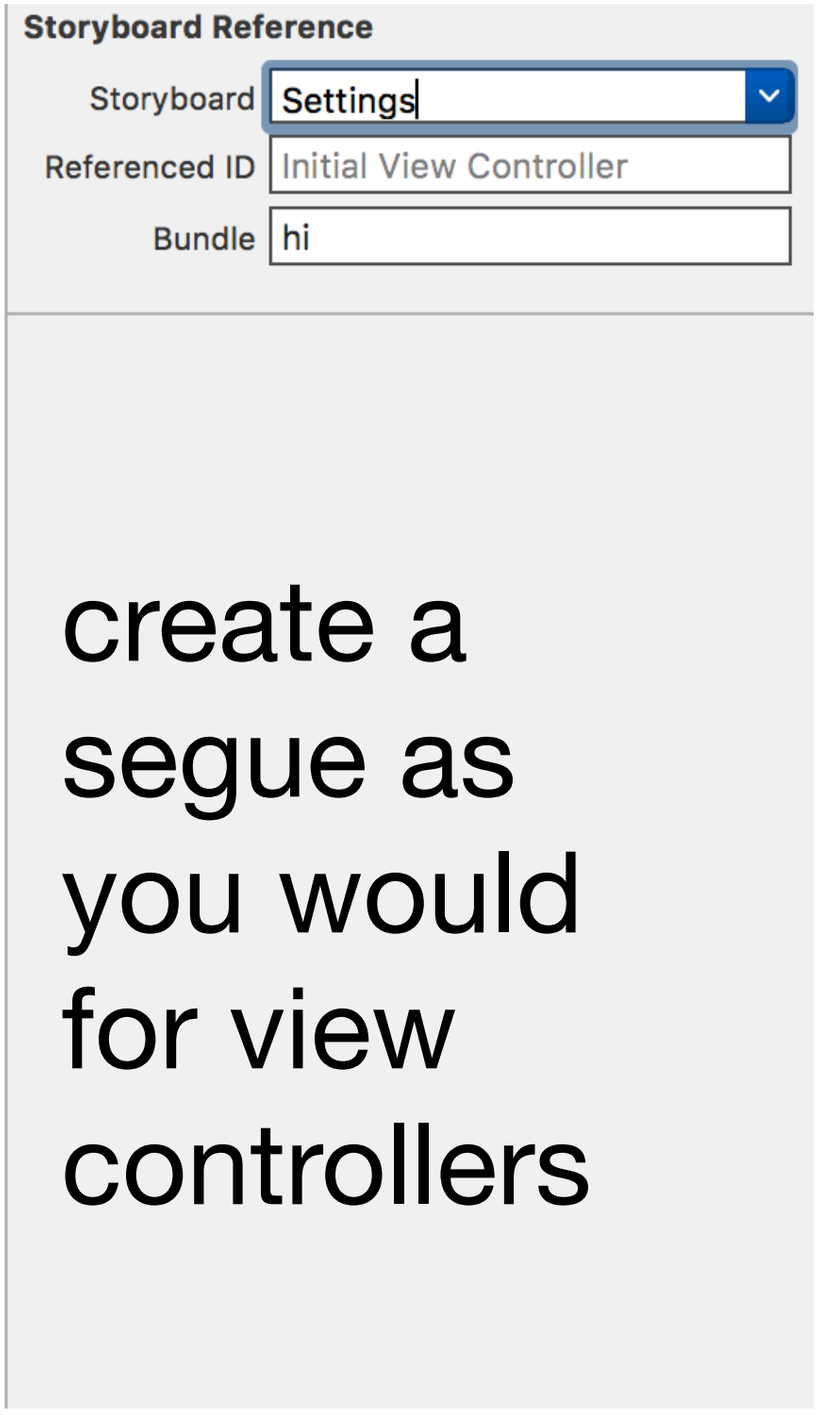


Create a Storyboard Reference



Create a Storyboard Reference





create a segue as you would for view controllers

Storyboards References

Storyboard references are powerful! You can create references to

- storyboards in different bundles,
- the same storyboard, and
- specific view controllers within other storyboards

Editor > Refactor Storyboard will automatically create a new storyboard file for selected view controllers

Programmatic Design

No Storyboard Needed

UI elements (buttons / labels / views) are instantiated in code and added as subviews

Pros

Better for version control

Scalable

Less limited

Cons

Steeper learning curve

Lots of boiler plate code

Not visual

Summary - iOS UI Implementation

Programmatic Design vs Storyboard

… so which one is better?

Depends on what you are creating

Often times, a combination of both may be the best solution

(i.e - instantiate views in storyboard, edit them programmatically)

Programmatic Design

Some important classes:

UIWindow - provides the backdrop for your app's content (usually only one per app)

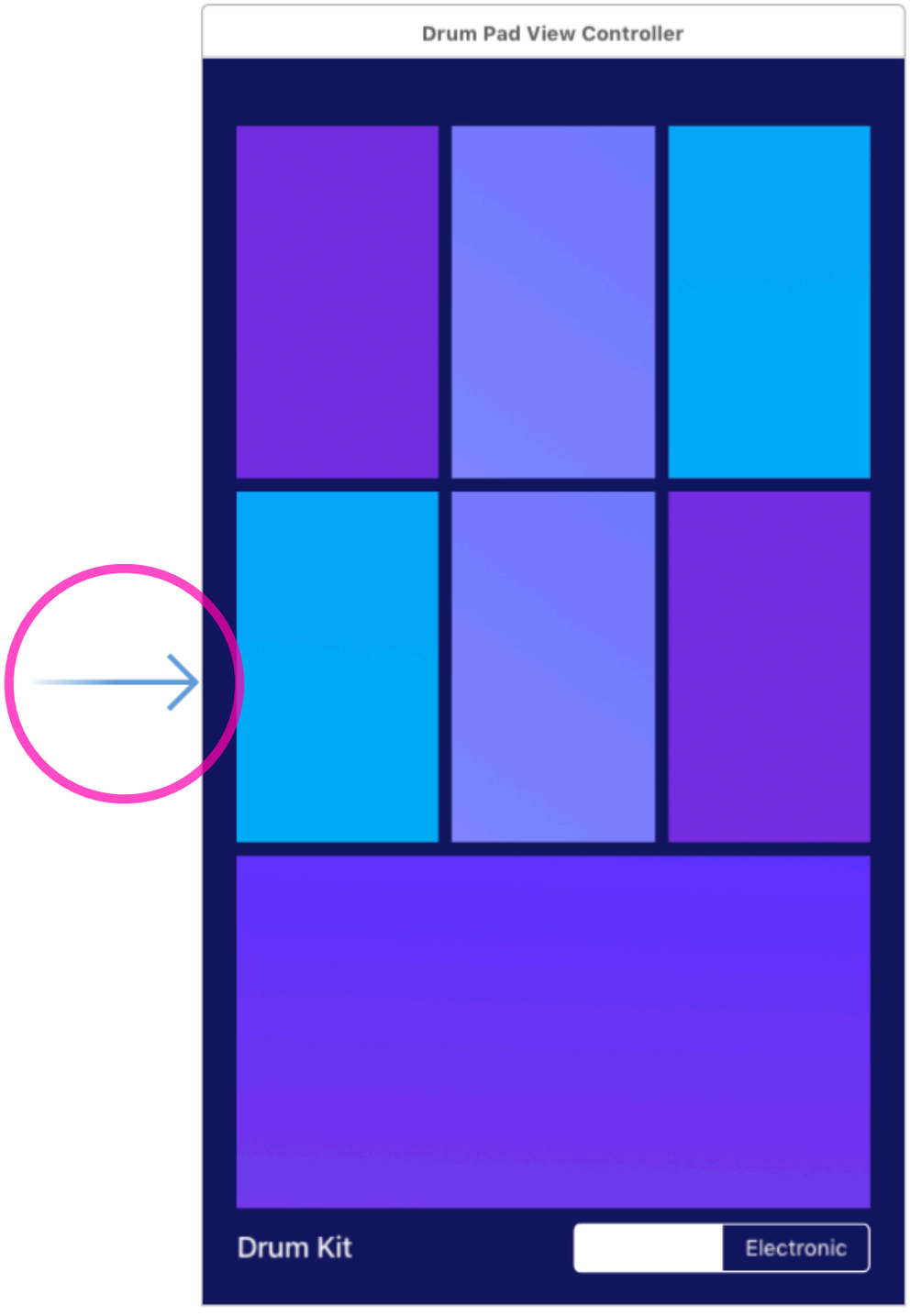
UIScreen - defines the properties of the user's device (get the bounds of user's device using `UIScreen.main.bounds`)

UIViewController - manages a set of UIView's

Programmatic Design (with no Storyboard)

To get rid of your storyboard, delete both the **Main.storyboard** file and it's reference in **Info.plist**

To programmatically set the initial view controller, you'll need to edit your **AppDelegate.swift.** This is equivalent to setting the "initial view controller" property in Storyboard (represented by the arrow icon)



Programmatic Design (with no Storyboard)

Found in AppDelegate.swift

```
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

 var window: UIWindow?

 func application(_ application: UIApplication,
         didFinishLaunchingWithOptions launchOptions:
         [UIApplicationLaunchOptionsKey: Any]?) -> Bool {

   // this code executes when your app is opened for the
              first time

   return true
  }
...
```

Setting your initial View Controller Programmatically (Example)

Programmatic Design (with no Storyboard)

Found in AppDelegate.swift

```
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

 var window: UIWindow?

  func application(_ application: UIApplication,
          didFinishLaunchingWithOptions launchOptions:
          [UIApplicationLaunchOptionsKey: Any]?) -> Bool {
    window = UIWindow(frame: UIScreen.main.bounds)
    let myViewController = MyViewController()
    window?.rootViewController = myViewController
    window?.makeKeyAndVisible()
    return true
  }
...
```

The window displays the app's content on the device's main screen.

Programmatic Design (with no Storyboard)

Found in AppDelegate.swift

```
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

 var window: UIWindow?

 func application(_ application: UIApplication,
        didFinishLaunchingWithOptions launchOptions:
        [UIApplicationLaunchOptionsKey: Any]?) -> Bool {
   window = UIWindow(frame: UIScreen.main.bounds)
   let myViewController = MyViewController()
   window?.rootViewController = myViewController
   window?.makeKeyAndVisible()
   return true
  }
...
```

Set the window to be
the size of the user's screen

Programmatic Design (with no Storyboard)

Found in AppDelegate.swift

```
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

 var window: UIWindow?

 func application(_ application: UIApplication,
        didFinishLaunchingWithOptions launchOptions:
        [UIApplicationLaunchOptionsKey: Any]?) -> Bool {
   window = UIWindow(frame: UIScreen.main.bounds)
   let myViewController = MyViewController()
   window?.rootViewController = myViewController
   window?.makeKeyAndVisible()
   return true
  }
...
```

Instantiate a View Controller to be the window's root view controller

Programmatic Design (with no Storyboard)

Found in AppDelegate.swift

```
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

 var window: UIWindow?

 func application(_ application: UIApplication,
        didFinishLaunchingWithOptions launchOptions:
        [UIApplicationLaunchOptionsKey: Any]?) -> Bool {
   window = UIWindow(frame: UIScreen.main.bounds)
   let myViewController = MyViewController()
   window?.rootViewController = myViewController
   window?.makeKeyAndVisible()
   return true
  }
...
```

Set the window's
root view controller property

Programmatic Design (with no Storyboard)

Found in AppDelegate.swift

```
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

 var window: UIWindow?

 func application(_ application: UIApplication,
        didFinishLaunchingWithOptions launchOptions:
        [UIApplicationLaunchOptionsKey: Any]?) -> Bool {
   window = UIWindow(frame: UIScreen.main.bounds)
   let myViewController = MyViewController()
   window?.rootViewController = myViewController
   window?.makeKeyAndVisible()
   return true
  }
...
```

Make the window visible to the user

Programmatic Design (with no Storyboard)

Found in AppDelegate.swift

```
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

 var window: UIWindow?

 func application(_ application: UIApplication,
        didFinishLaunchingWithOptions launchOptions:
        [UIApplicationLaunchOptionsKey: Any]?) -> Bool {
   window = UIWindow(frame: UIScreen.main.bounds)
   let myViewController = MyViewController()
   window?.rootViewController = myViewController
   window?.makeKeyAndVisible()
   return true
  }
...
```

Now the user will see “myViewController” upon opening this application

Programmatic Design

To create UI elements programmatically, you'll need to do the following:

1. Instantiate the UI element

    i.e. `let myButton = UIButton()`

2. Add the view as a subview to your superview using `addSubview`

    i.e. `superview.addSubview(myButton)`

3. Set the position and size of your view either using **frames** or **layout constraints**

Programmatic Design : Example

Suppose we wanted to add a button to our view

in Storyboard

Drag and drop a UIButton into your storyboard from the **Object Library**

Customize using **Attributes Inspector**

Setup Constraints

Programmatically

```
let myBtn = UIButton(frame:
                    CGRect(x: 50,
                           y: 100,
                       width: 200,
                      height: 100))

myBtn.setTitle("Click me!”,
                        for: .normal)
myBtn.backgroundColor = .red
view.addSubview(myBtn)
```

Views and Geometry

Views / UIView

The `UIView` class defines a rectangular area on your user's screen

This area can be used for managing content, holding other views, registering touch events, etc.

Classes like `UIImageView` and `UILabel` are special types of `UIView's` (they both subclass `UIView`)

CGRect and CGPoint

CGRect - defines position and size

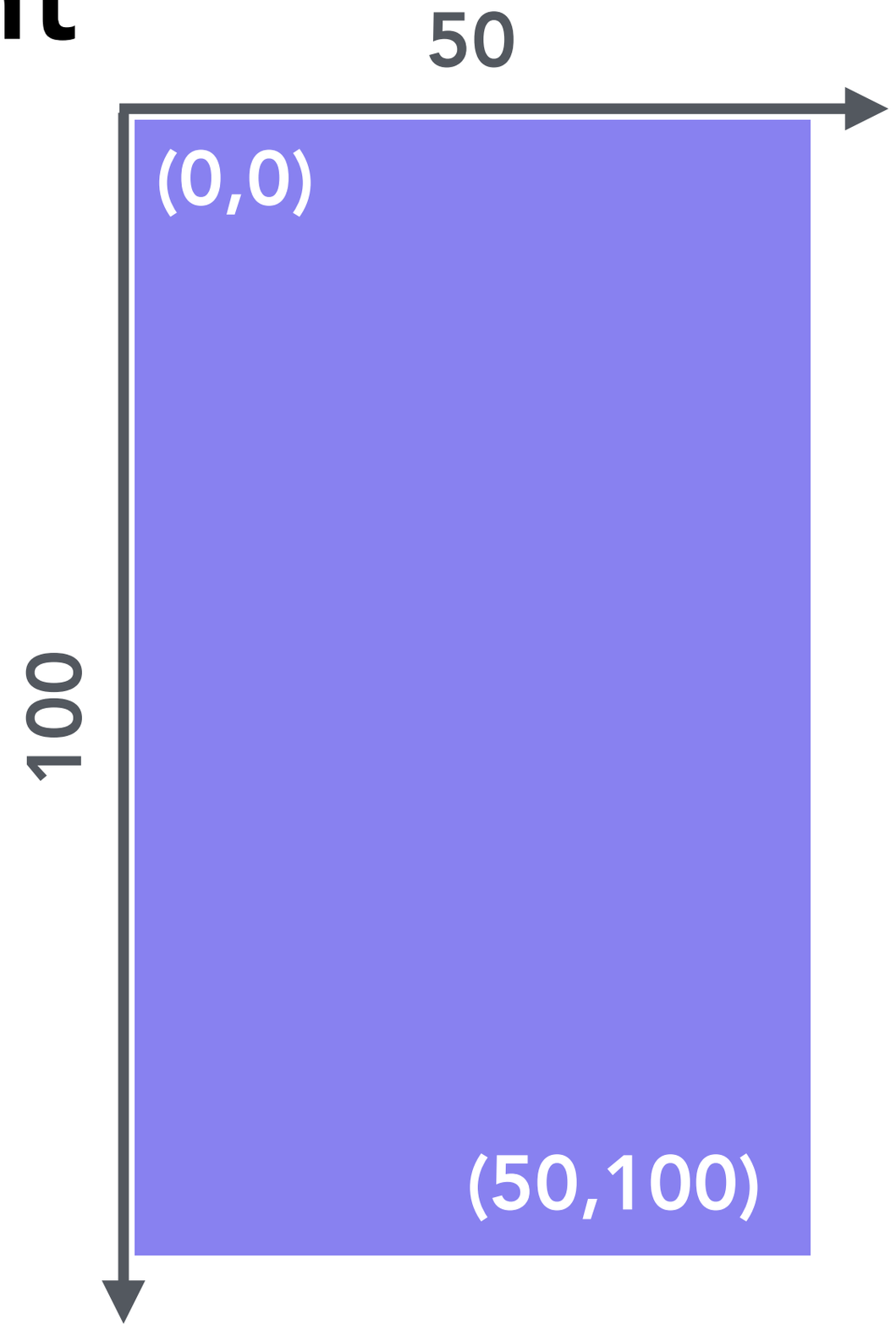
```
CGRect(x: 0, y: 0,
       width: 100,
       height: 100)
```

CGPoint - defines a position

```
CGPoint(x: 0, y: 0)
```

CGSize - defines a size

```
CGSize(width: 100,
       height: 100)
```



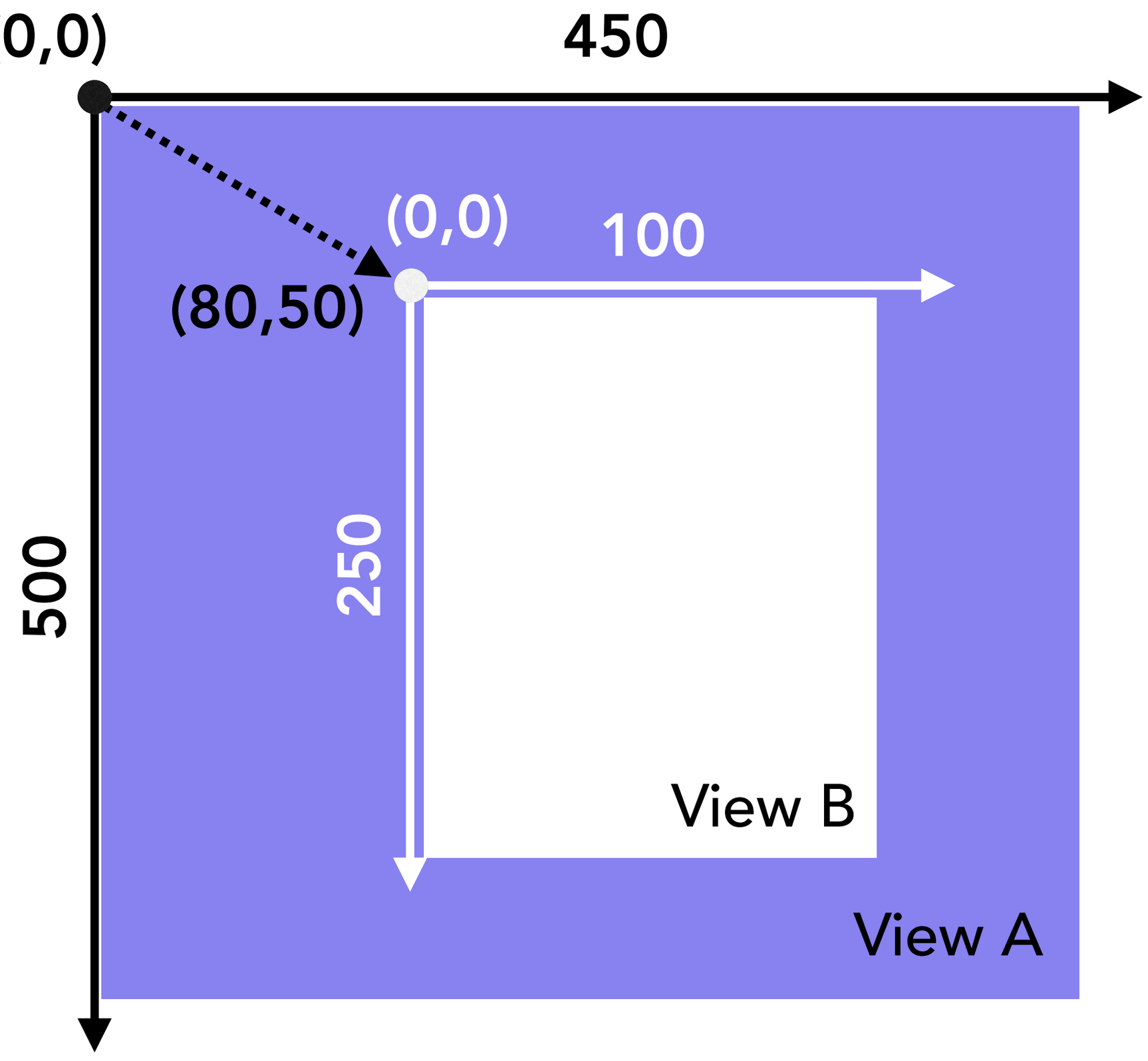
UIView : Geometry

A `UIView's` geometry is defined by the view's `frame`, `bounds`, and `center` properties

`frame: CGFrame` - the coordinates and dimensions of the view **in the coordinate system of its superview**

`bounds: CGFrame` - the coordinates and dimensions of the view **relative to itself**

`center: CGPoint` - the center of the view (used for positioning of the view)



View A frame

x, y = (?,?)
width = 450
height = 500

View A bounds

x, y = (?,?)
width = 450
height = 500

View B frame

x, y = (?,?)
width = 100
height = 250

View B bounds

x, y = (?,?)
width = 100
height = 250

`frame:` uses the **coordinate system of its superview**

`bounds:` uses coordinates **relative to itself**

Positioning / Sizing Views

Two ways of setting the size and position of your views programmatically

1. Using **frames** / **bounds** (`initWithFrame CGRect, CGPoint)`

2. Using **AutoLayout** (`NSLayoutConstraints`)

Positioning and Sizing Views Using Frames

```
let myFrame = CGRect(x: 0, y: 0,
 width: UIScreen.main.bounds.width - 16,
 height: 100)

let myButton = UIButton(frame: myFrame)

myButton.center = view.center

view.addSubview(myButton)
```

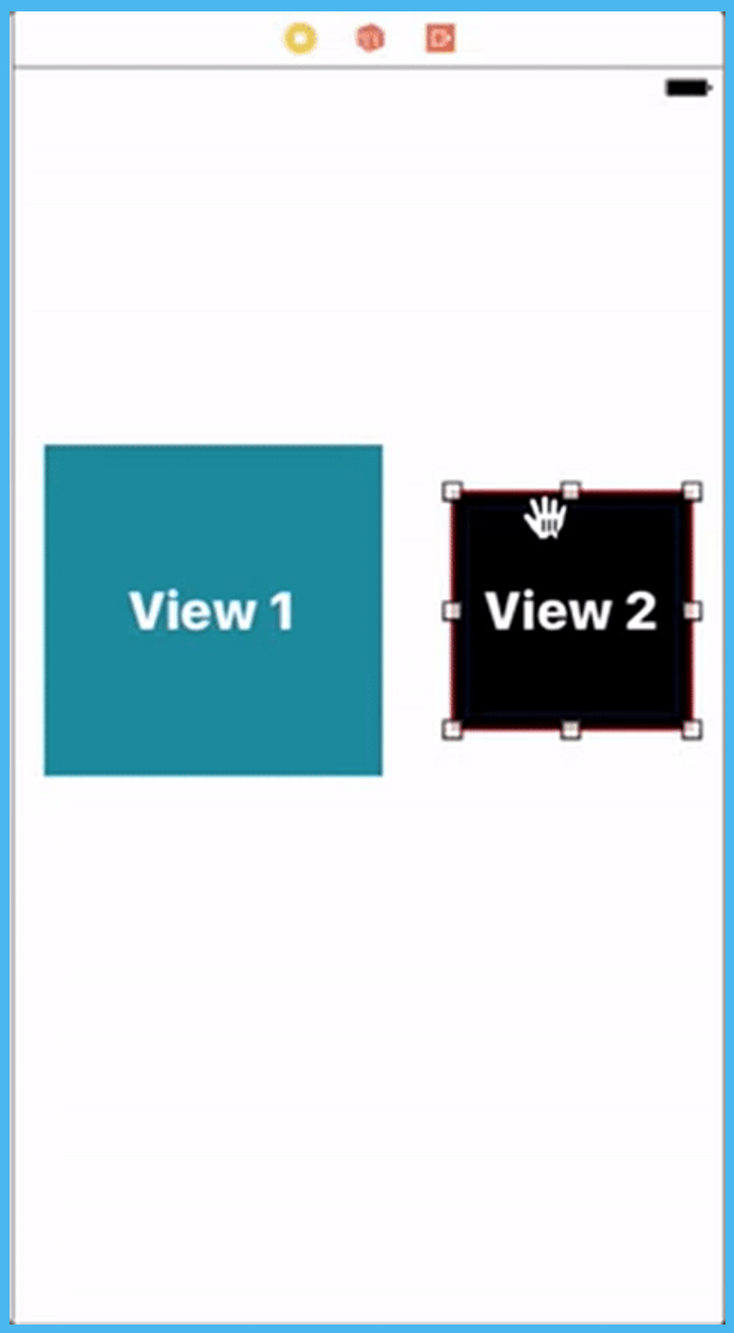
Positioning and Sizing Views with AutoLayout

```
btn.translatesAutoresizingMaskIntoConstraints = false

// constraints to center the button horizontally in the view
let myConstraints = [
    btn.centerXAnchor.constraint(equalTo: view.centerXAnchor),
    btn.centerYAnchor.constraint(equalTo: view.centerYAnchor),
    btn.leadingAnchor.constraint(equalTo: view.leadingAnchor,
                                 constant: 8),
     btn.trailingAnchor.constraint(equalTo: view.trailingAnchor,
                                   constant: 8),
     btn.heightAnchor.constraint(equalToConstant: 100)
   ]

NSLayoutConstraint.activate(myConstraints)
```

In this example, we create a list of constraints, then batch activate them (rather than doing it one by one)

Programmatic AutoLayout

Layout Anchors



Equivalent
Storyboard Example

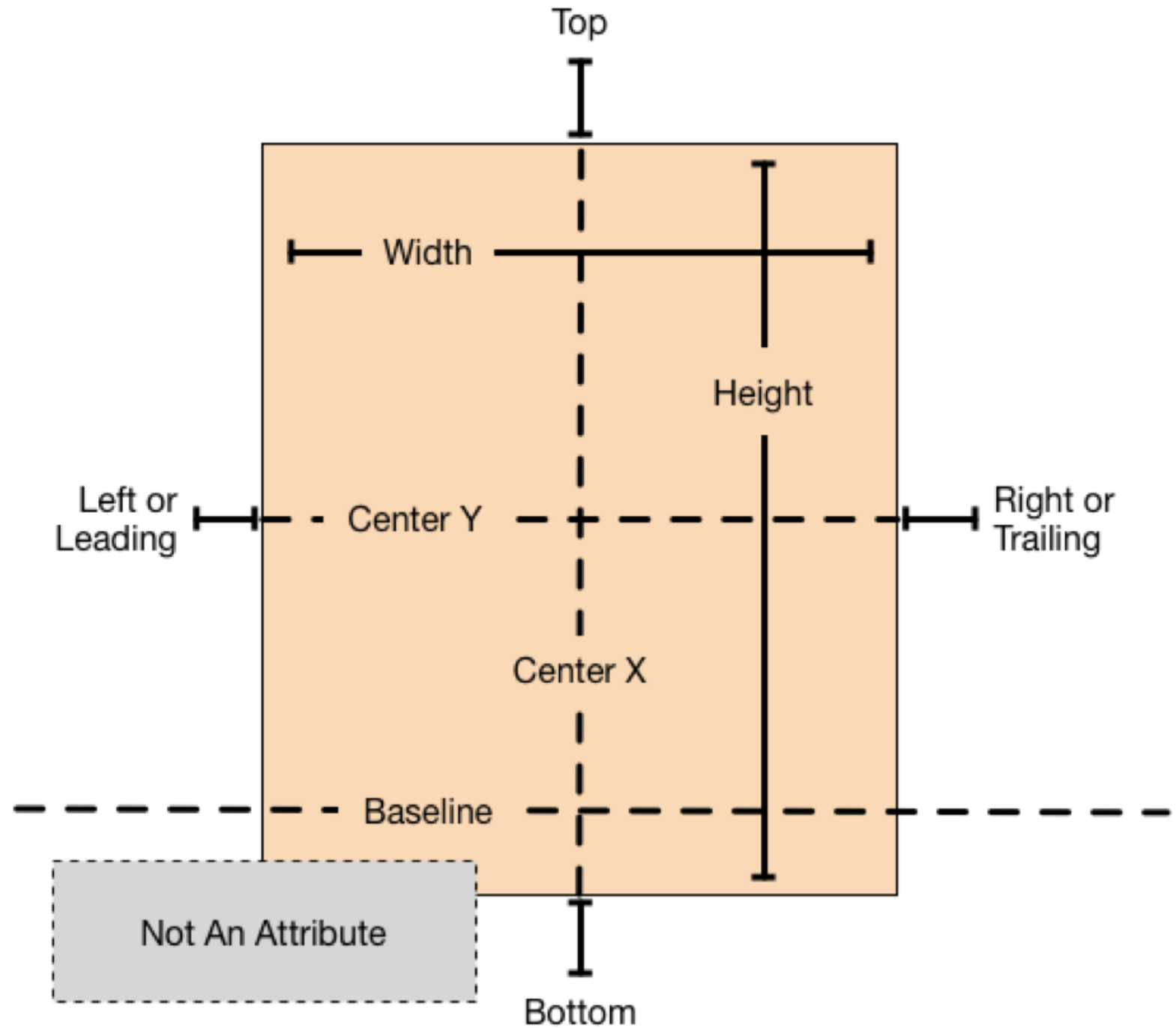
```
let constraint =
      view2.leadingAnchor.constraint(
      equalTo: view1.trailingAnchor,
      constant: 8)

constraint.isActive = true
```

In both of these examples,
the spacing between
views is set to 8 points

Programmatic AutoLayout

Layout Anchor Properties

Use these properties to create relationships between views



Presenting view controllers

To present new view controllers (i.e. show segue)

```
func present(_ viewControllerToPresent:
                        UIViewController,
                animated flag: Bool,
          completion: (() -> Void)? = nil)
```

Presenting VC's (with nav controllers)

To add vc's to a navigation controller stack

- instantiate a navigation controller
- add viewcontrollers to the nav controllers "viewcontrollers" array or set a root view controller
- navigate by "pushing" view controllers, or popping to the root view controller

Presenting VC's (with nav controllers)

adding a view controller to the stack (push)

```
let nextVC = ViewController2()
navigationController?.pushViewController(
                        nextVC, animated: true)
```

Note: navigationController will be nil if the current view controller is not within it's stack

Presenting VC's (with nav controllers)

pop (go back) to the root of you navigation controller

```
navigationController?.popToRootViewController
(animated: true)
```

Presenting VC's (with nav controllers)

pop (go back) to a different view controller, that isn't the root

```
navigationController?.popToViewController(pre
vVC, animated: true)
```

note: the VC you are navigating to must already be in the nav controller's stack

Demo

https://github.com/paigeplan/lec8