



# lecture 10

## UI/UX and Programmatic Design

cs198-001 : spring 2018

# Announcements

- custom app progress form due before lab (~1 minute)
  - will be released after lecture
- only 2 labs left (both very important)
- final showcase: Friday of RRR (5/4)  
4-6pm in HP Auditorium
  - If you have a conflict, let us know on Piazza now

# **Wednesday's Lab**

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

# **Mobile Machine Learning**

**Sp18 Machine Learning at Berkeley Workshop Series**

**April 18, 2018 | Wozniak Lounge | 8-10PM**

# Office Hours @Soda 346



# iOS Human Interface Guidelines

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

The image shows three iPhone screens side-by-side, illustrating various aspects of the iOS Human Interface Guidelines:

- Left Screen (Overview):** Shows the navigation bar with "Overview" and a dropdown arrow. Below it is a sidebar menu with the following items:
  - Design Principles
  - What's New in iOS 10
  - Interface Essentials
  - Interaction
  - Features
  - Visual Design
  - Graphics
  - UI Bars
  - UI Views
  - UI Controls
  - Extensions
  - Technologies
  - Resources
- Middle Screen:** Shows the iOS home screen with various app icons like Mail, Calendar, Photos, Camera, etc. It also displays the date (Monday, June 13), time (9:41 AM), and battery level (100%).
- Right Screen:** Shows a Siri suggestion card for a Lyft ride to SFO. The card includes the text "Get me a Lyft to SFO" and "tap to edit". It also displays a map showing the route from San Francisco International Airport to the user's current location.

# iOS Human Interface Guidelines

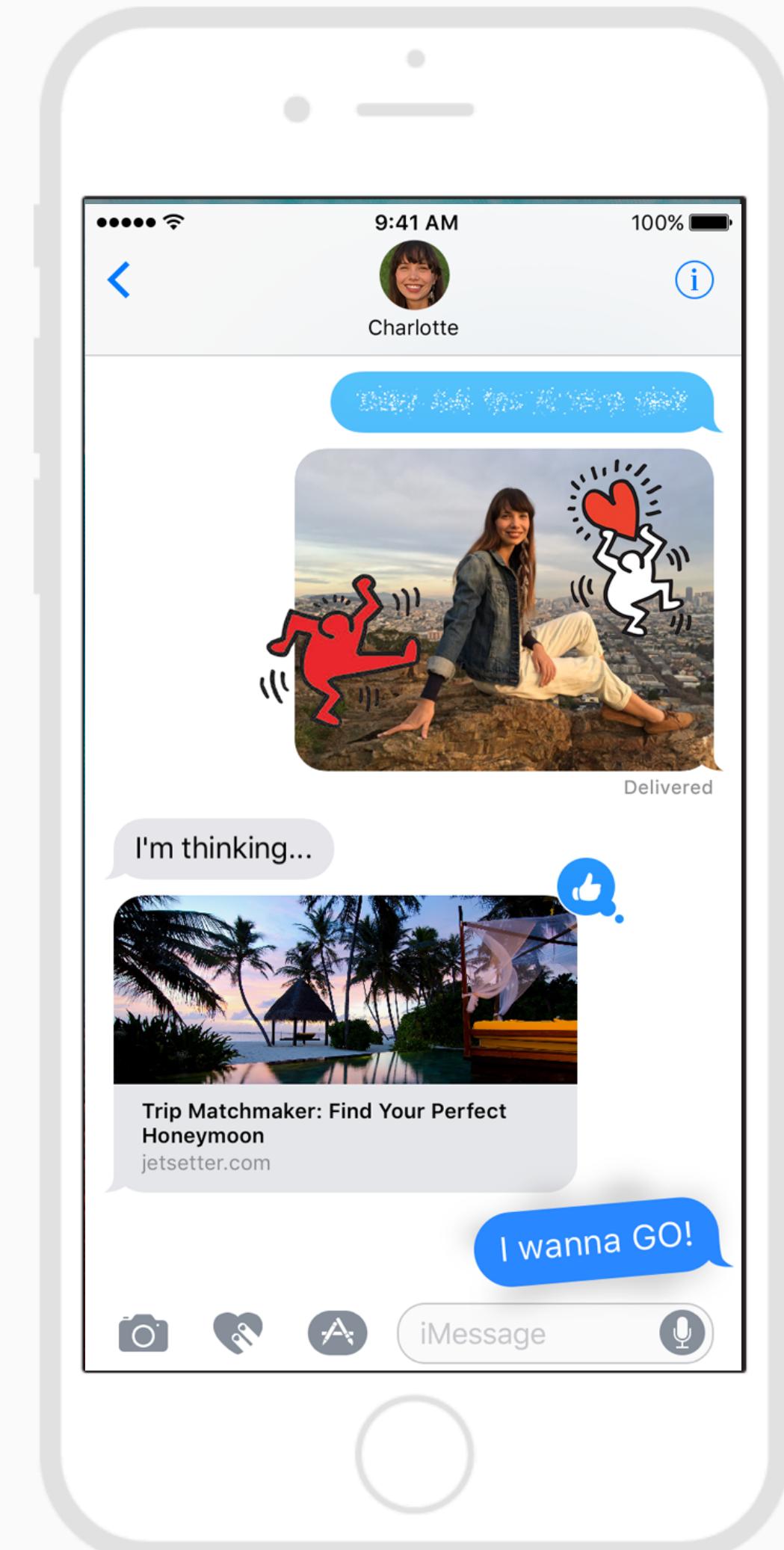
## iMessage Integration

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

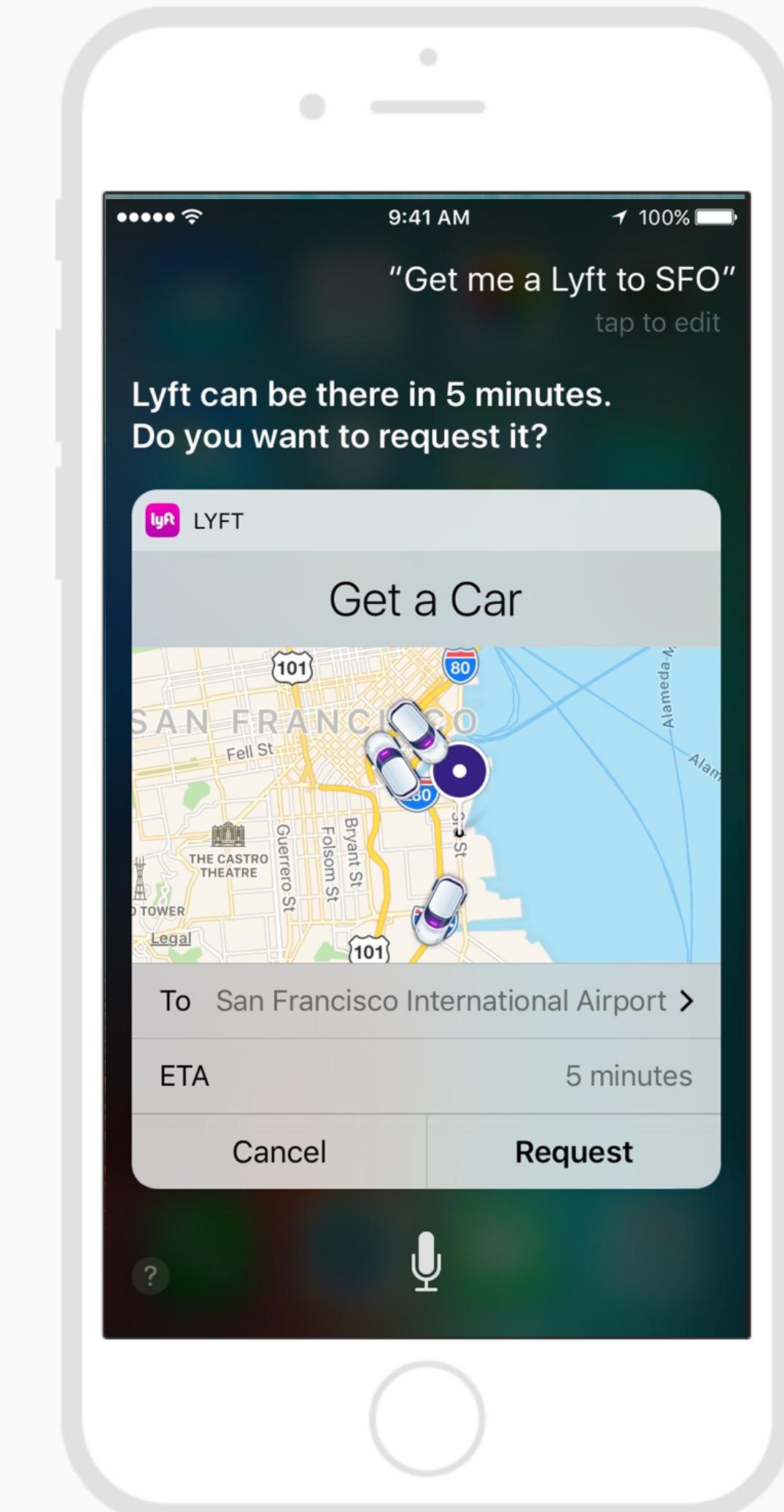
## Integration with Siri

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)

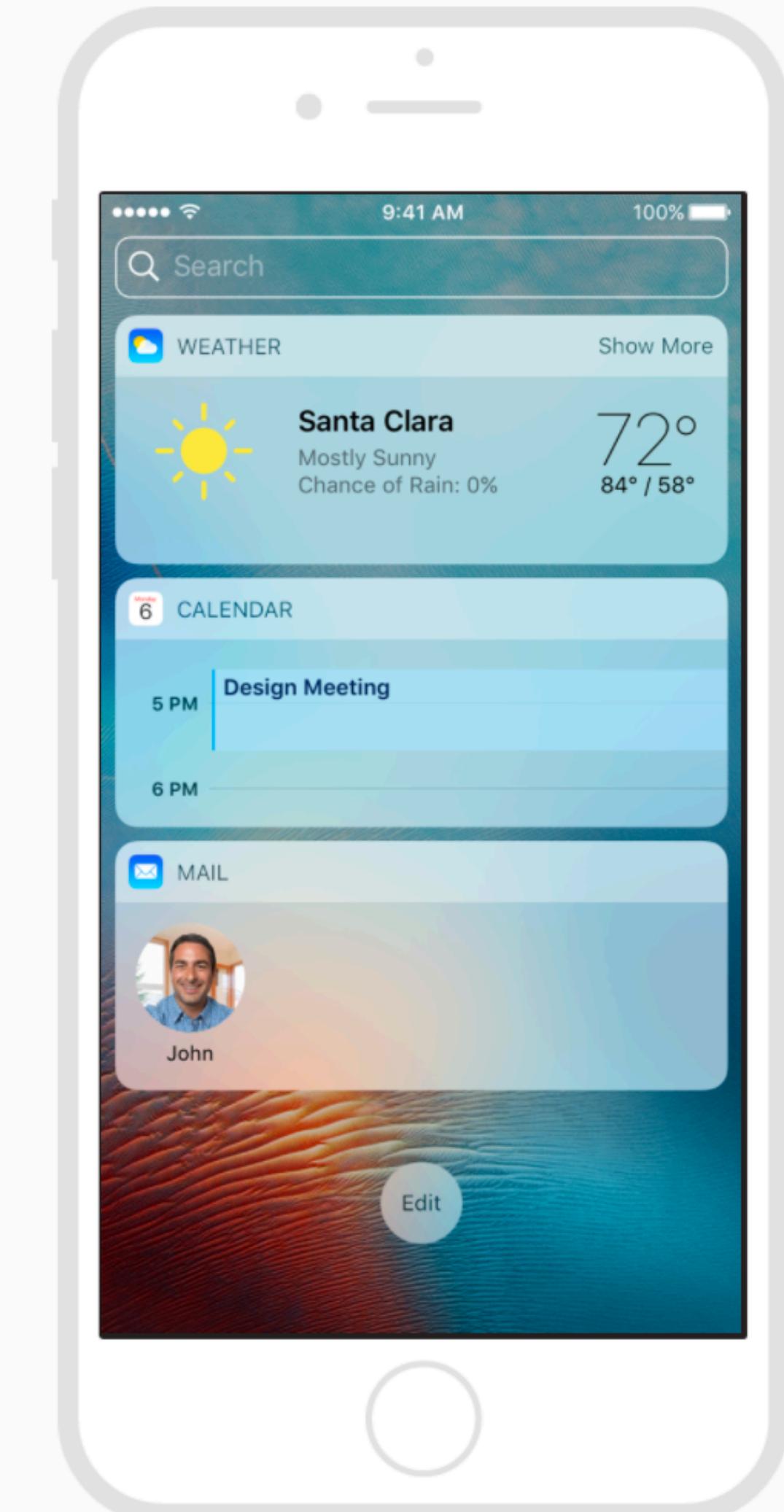


# iOS Human Interface Guidelines

## Search Widgets

Display notifications from your application on the user's Search and Home Screen

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



# iOS Human Interface Guidelines

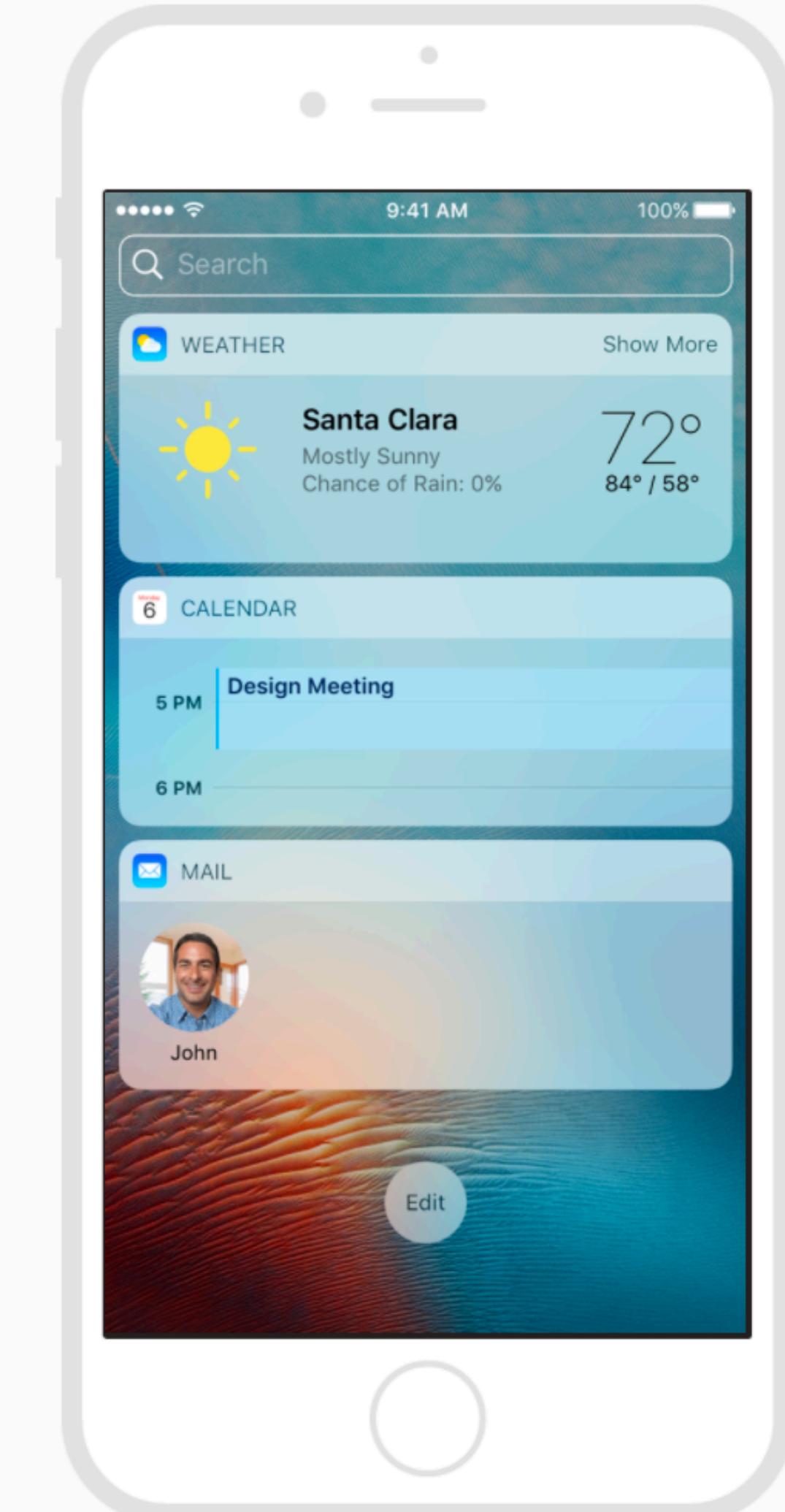
## Search Widgets

Display notifications from your application on the user's Search and 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

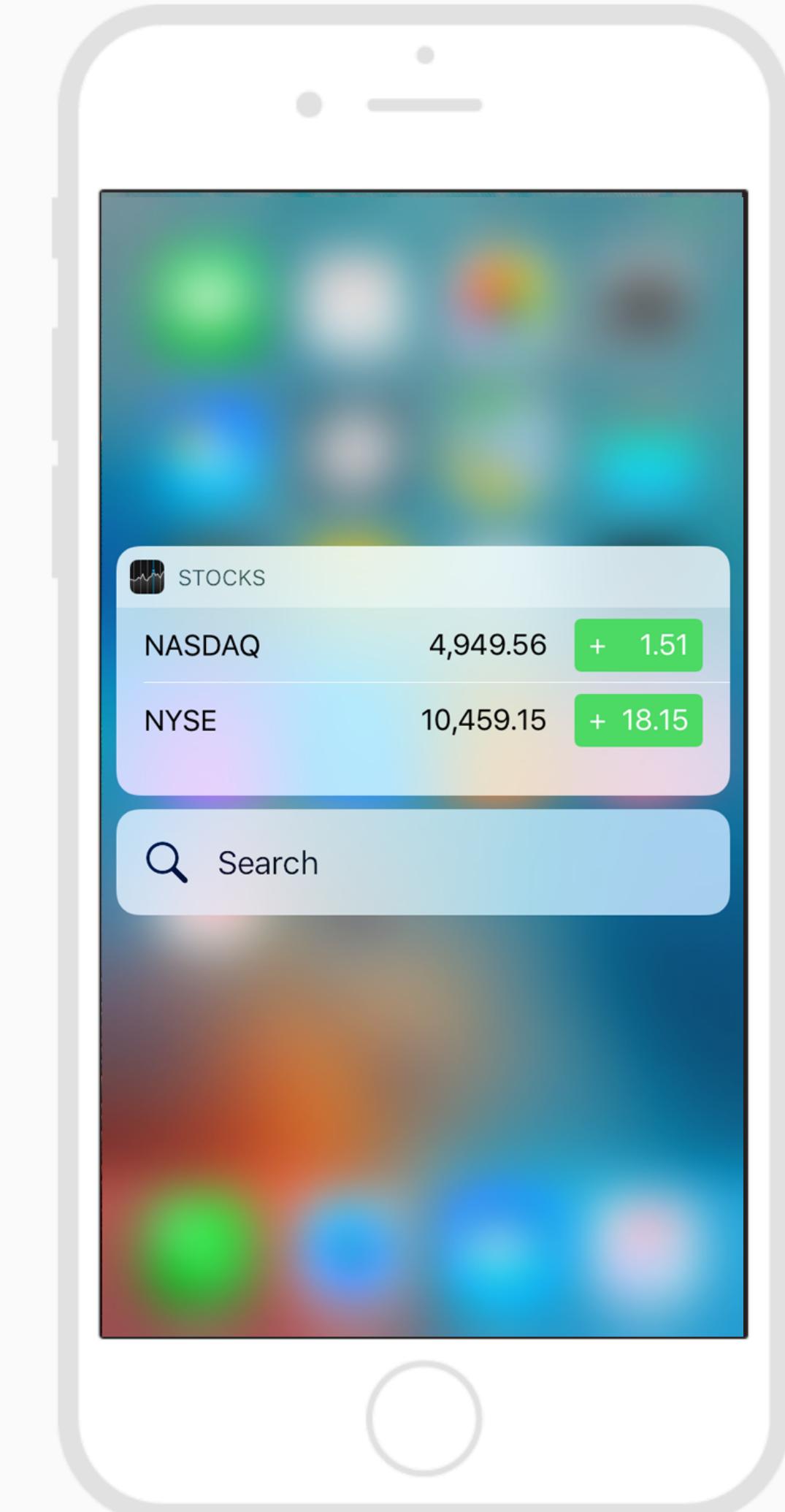
## Search Widgets

Display notifications from your application on the user's Search and Home Screen

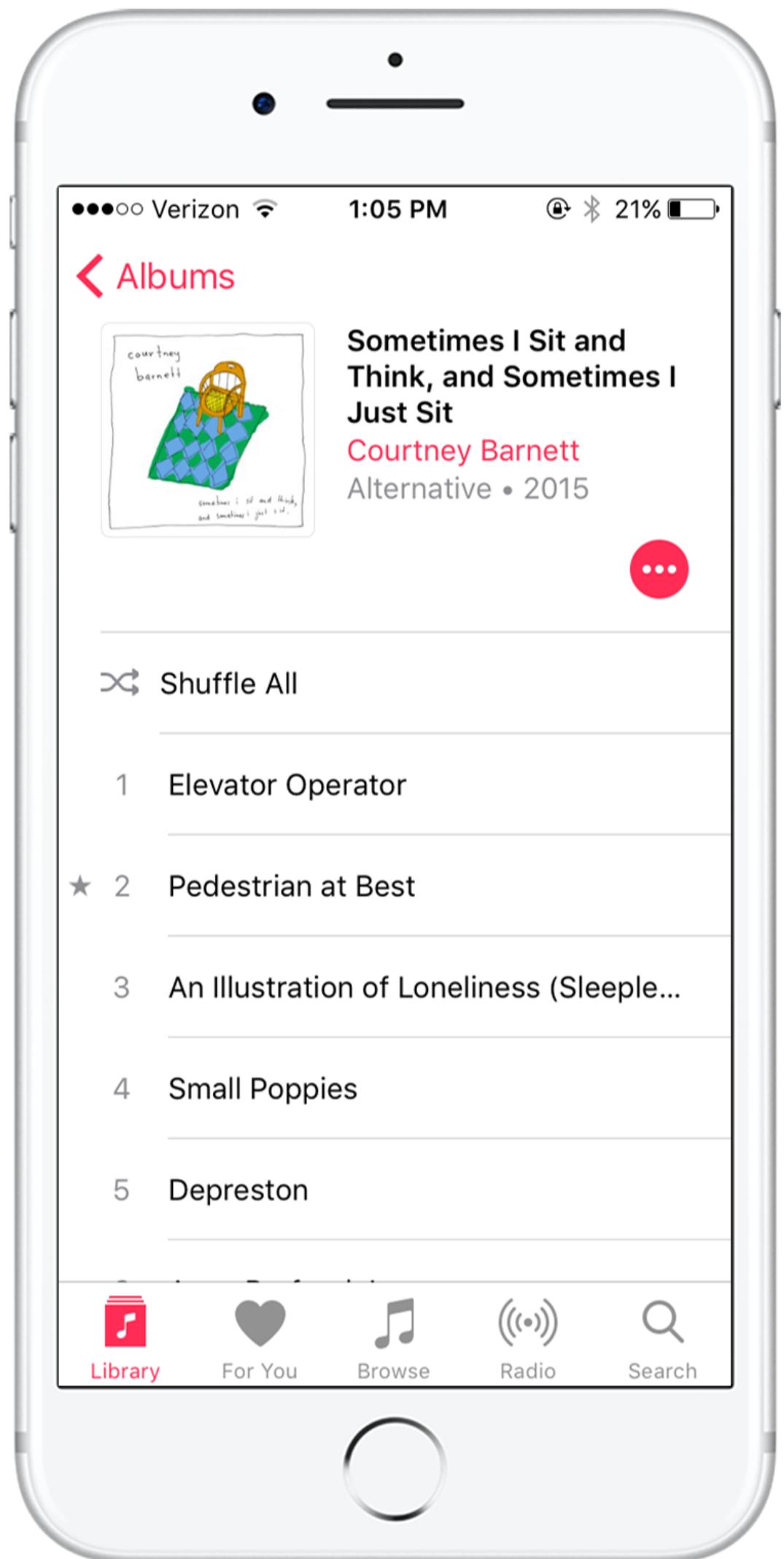
Very customizable (can add buttons, images, layout customization, etc.)

### 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

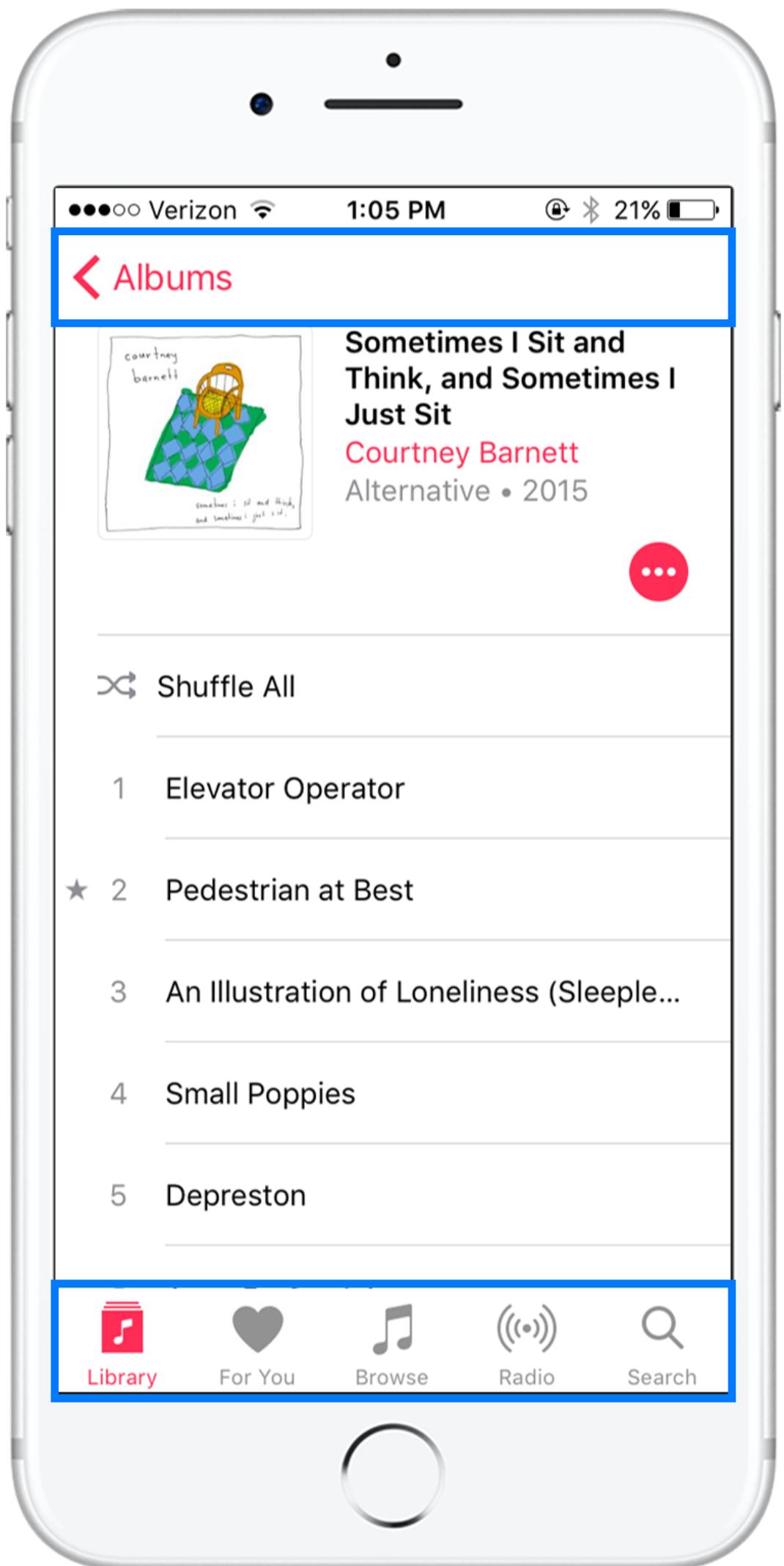
## Views

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

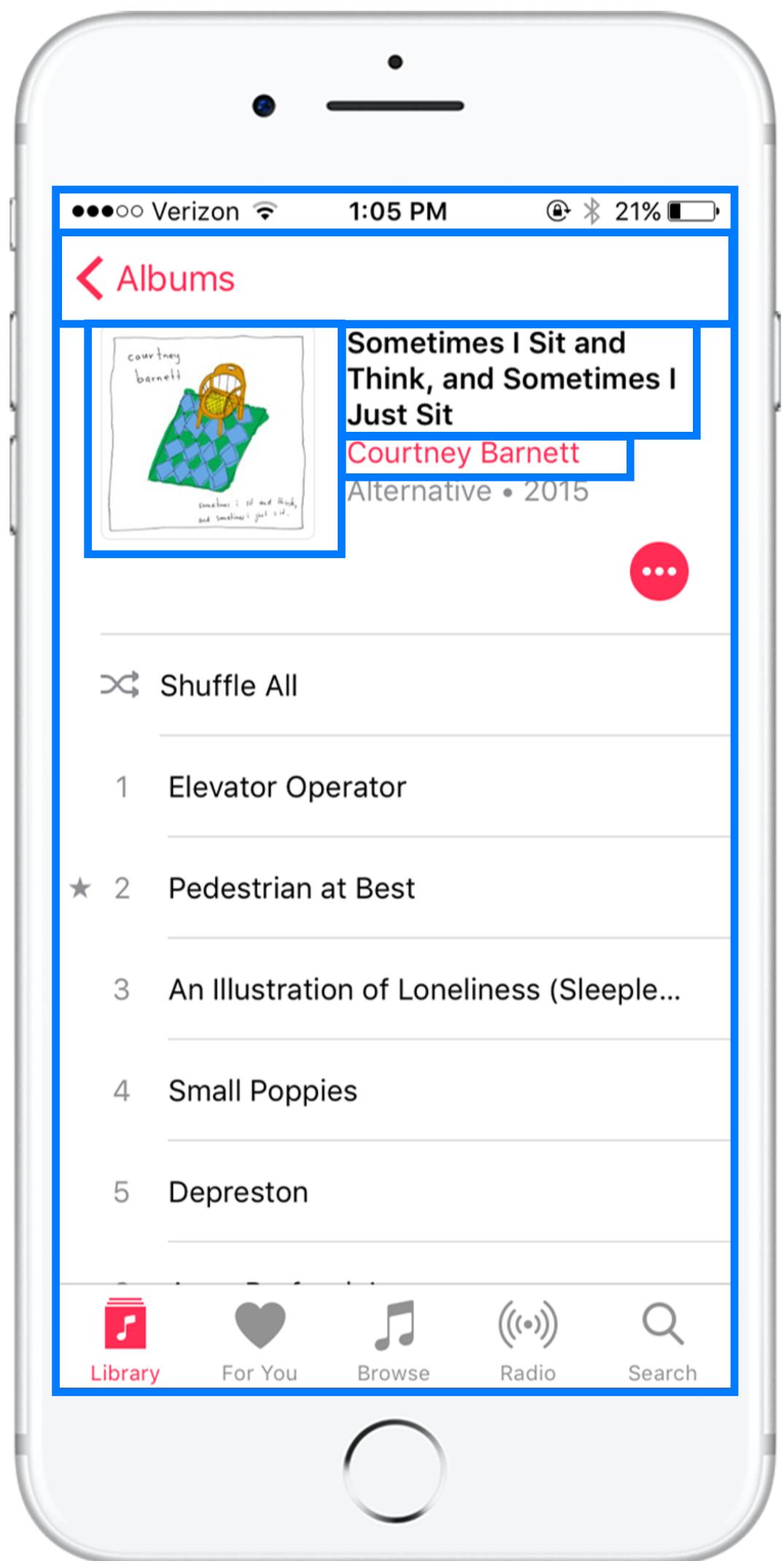
## Views

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

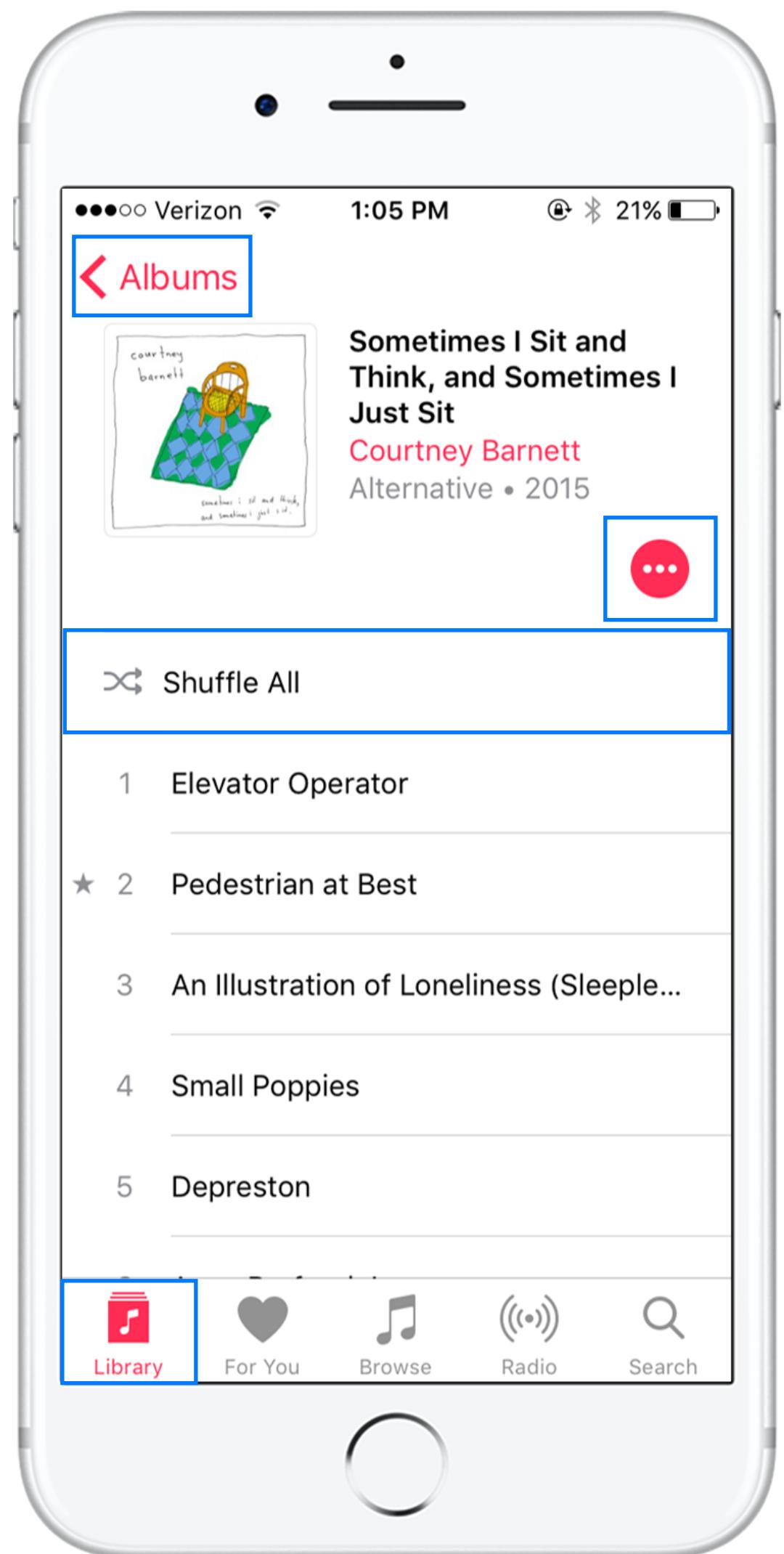
## Views

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

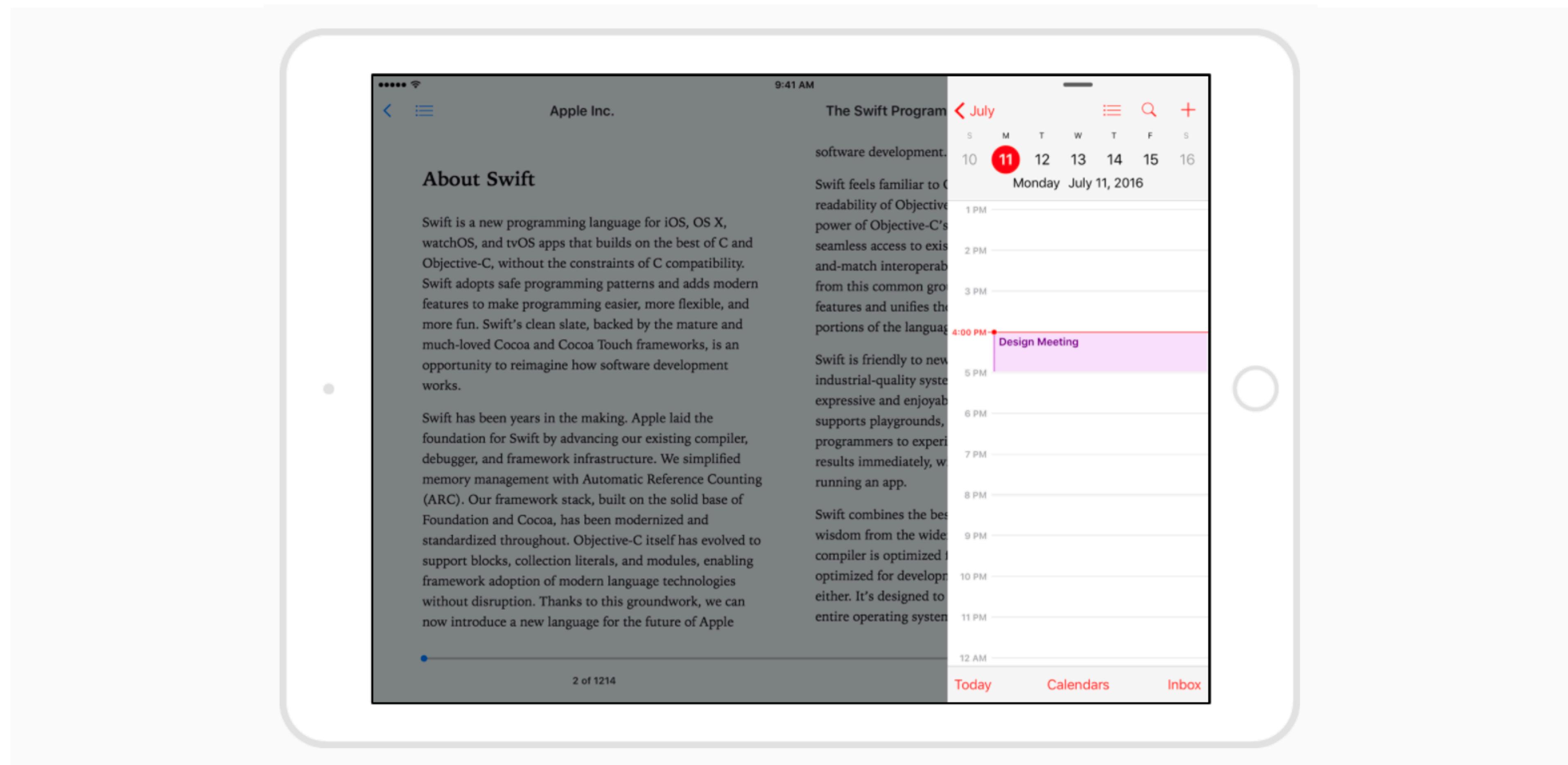
## Views

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

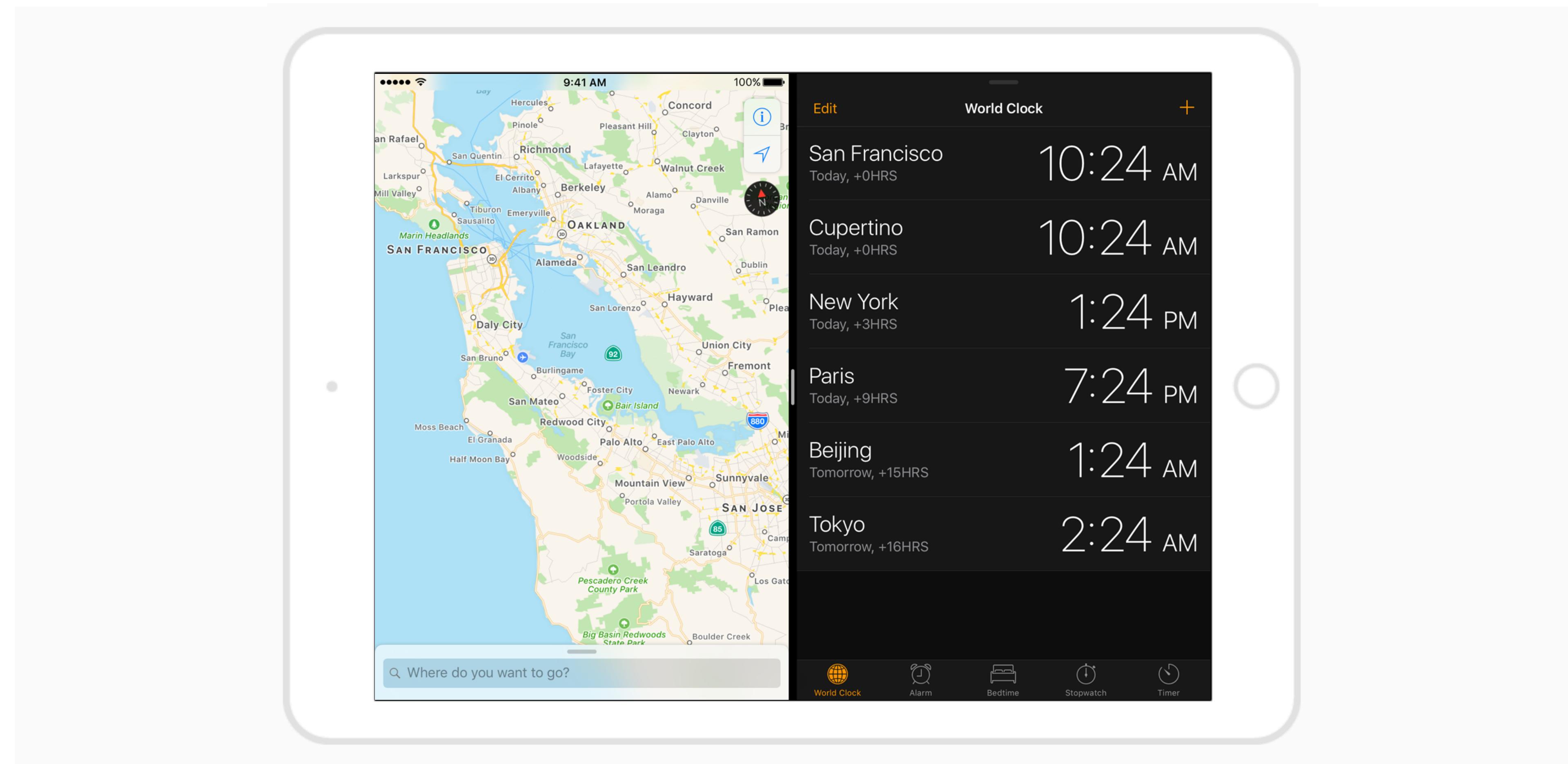
Buttons, text fields, segmented controls, etc.

# 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

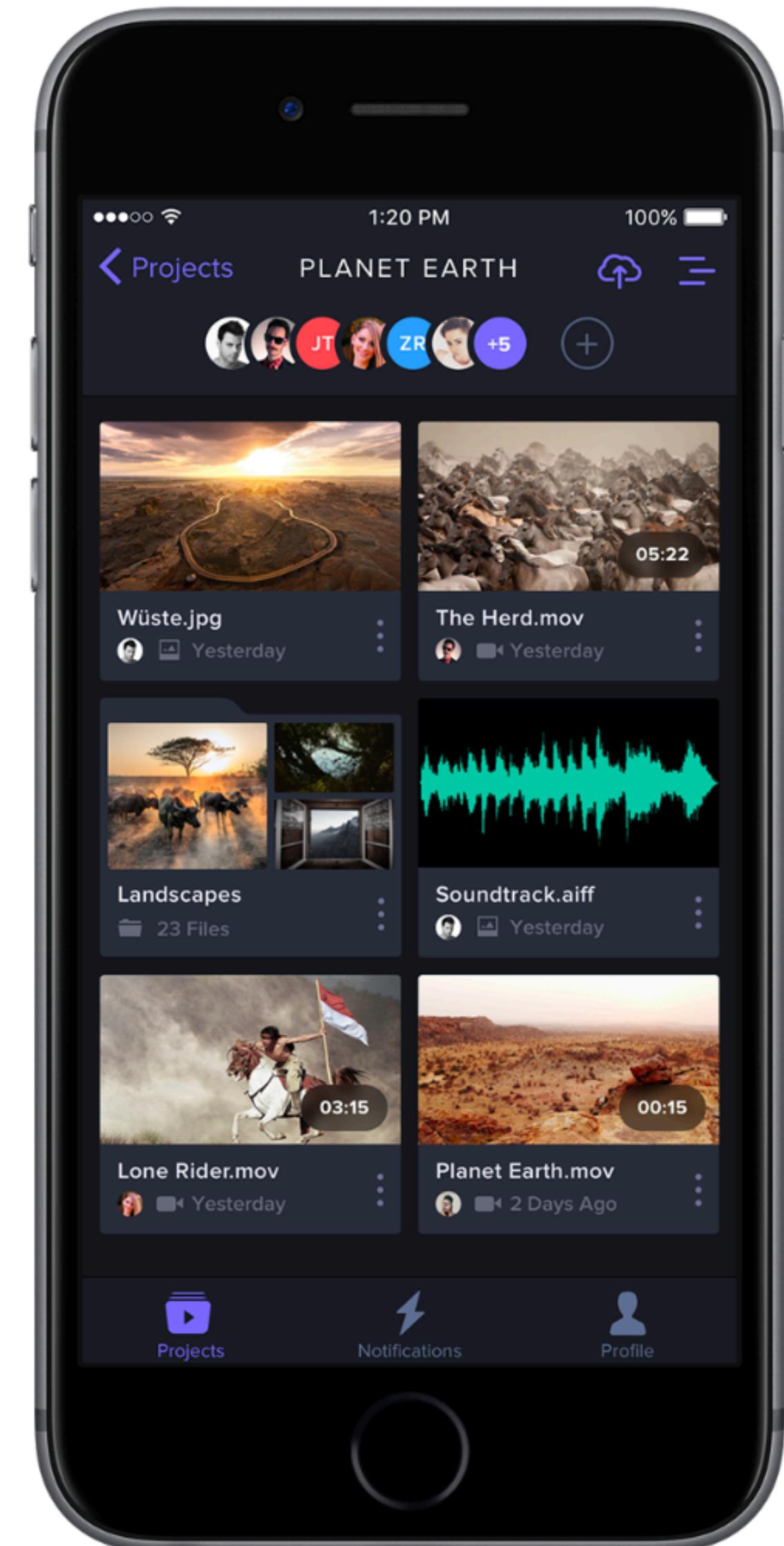
While it is important to have an app “image” or “brand”, avoid over-using logos, icon images, etc.

## 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

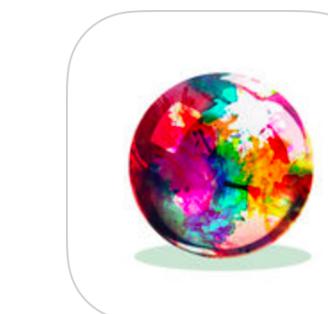
While it is important to have an app “image” or “brand”, avoid over-using logos, icon images, etc.

## 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



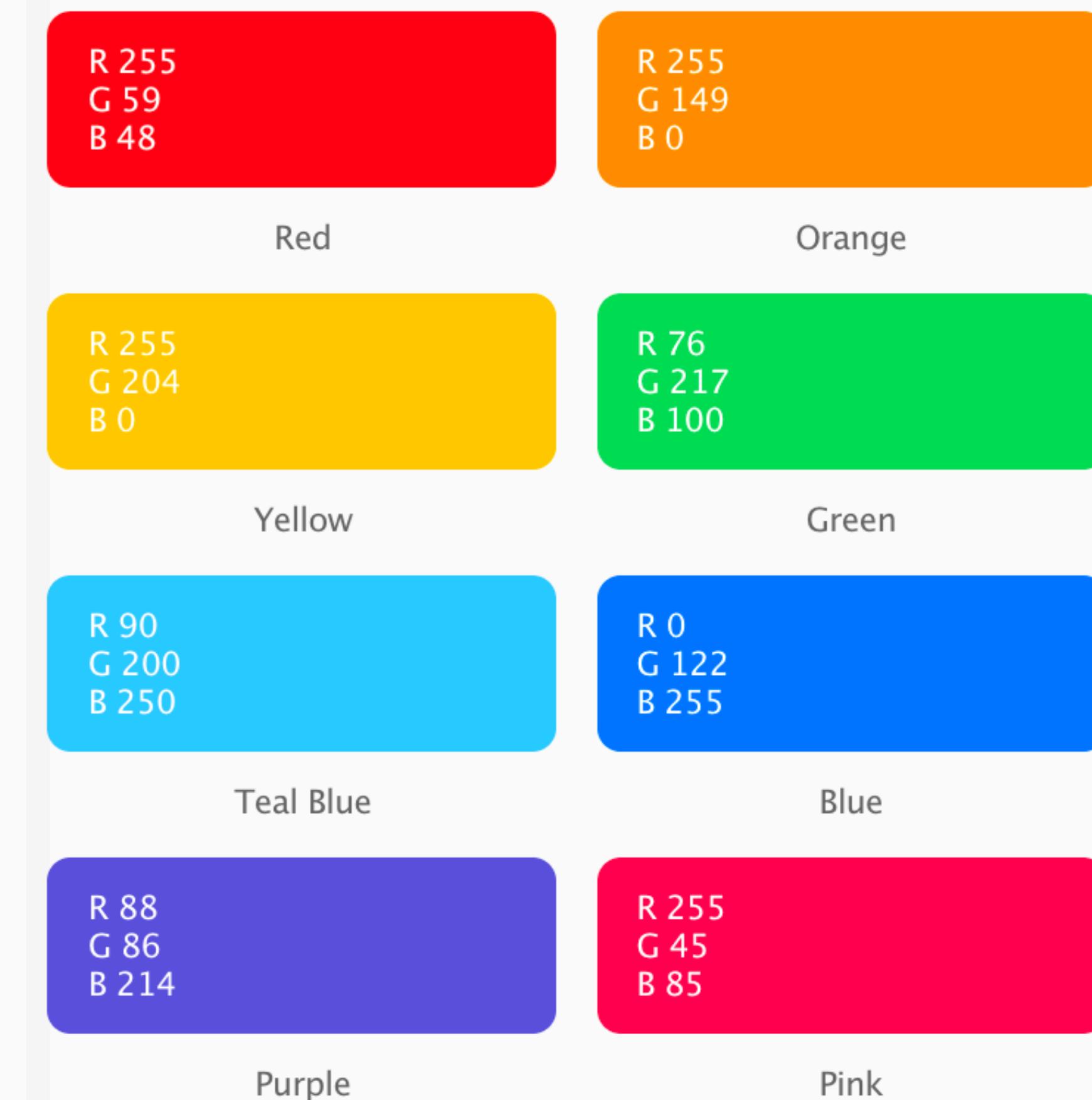
**INKS**  
State of Play Games

# 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

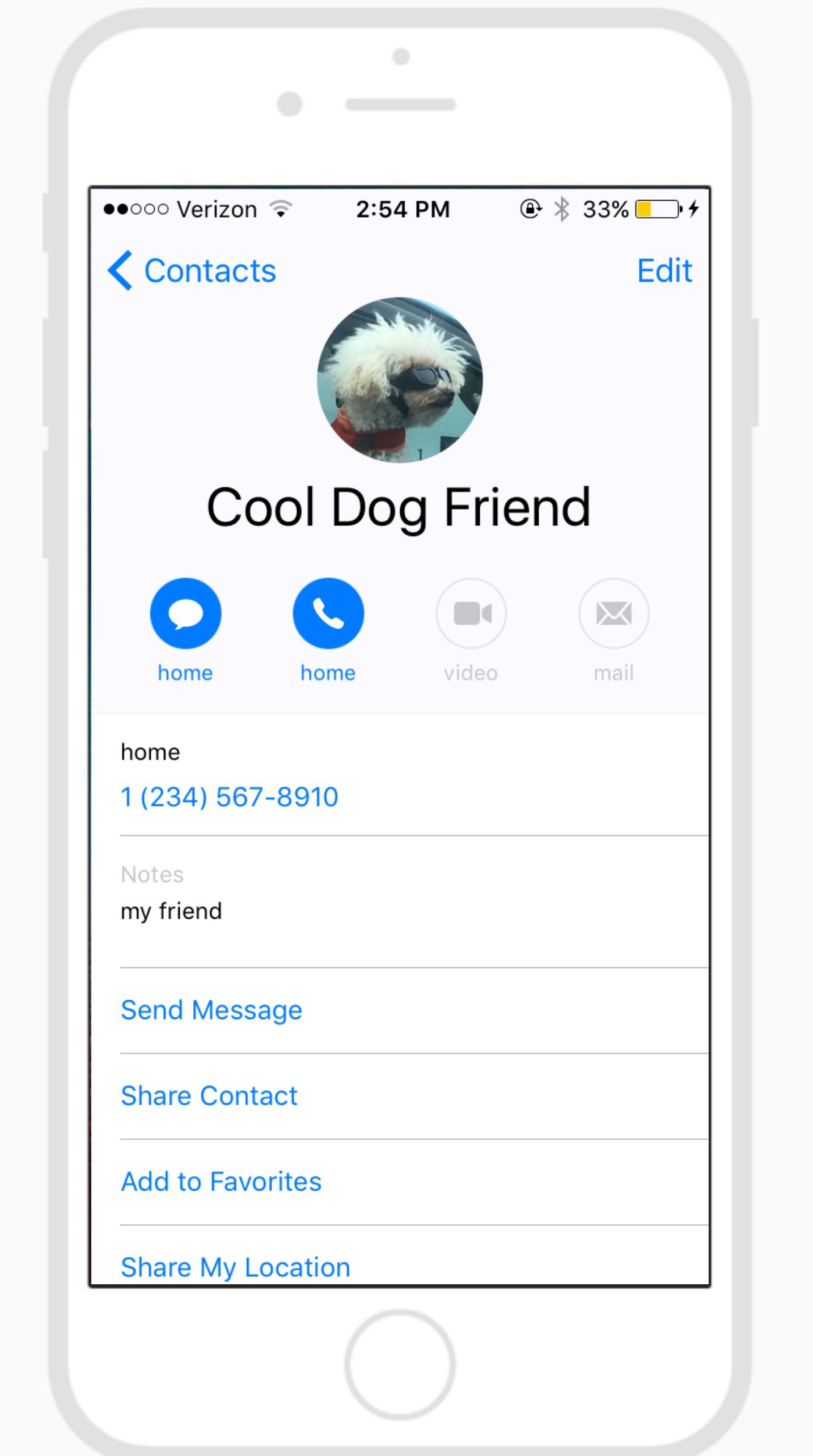


# iOS HIG : Color

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

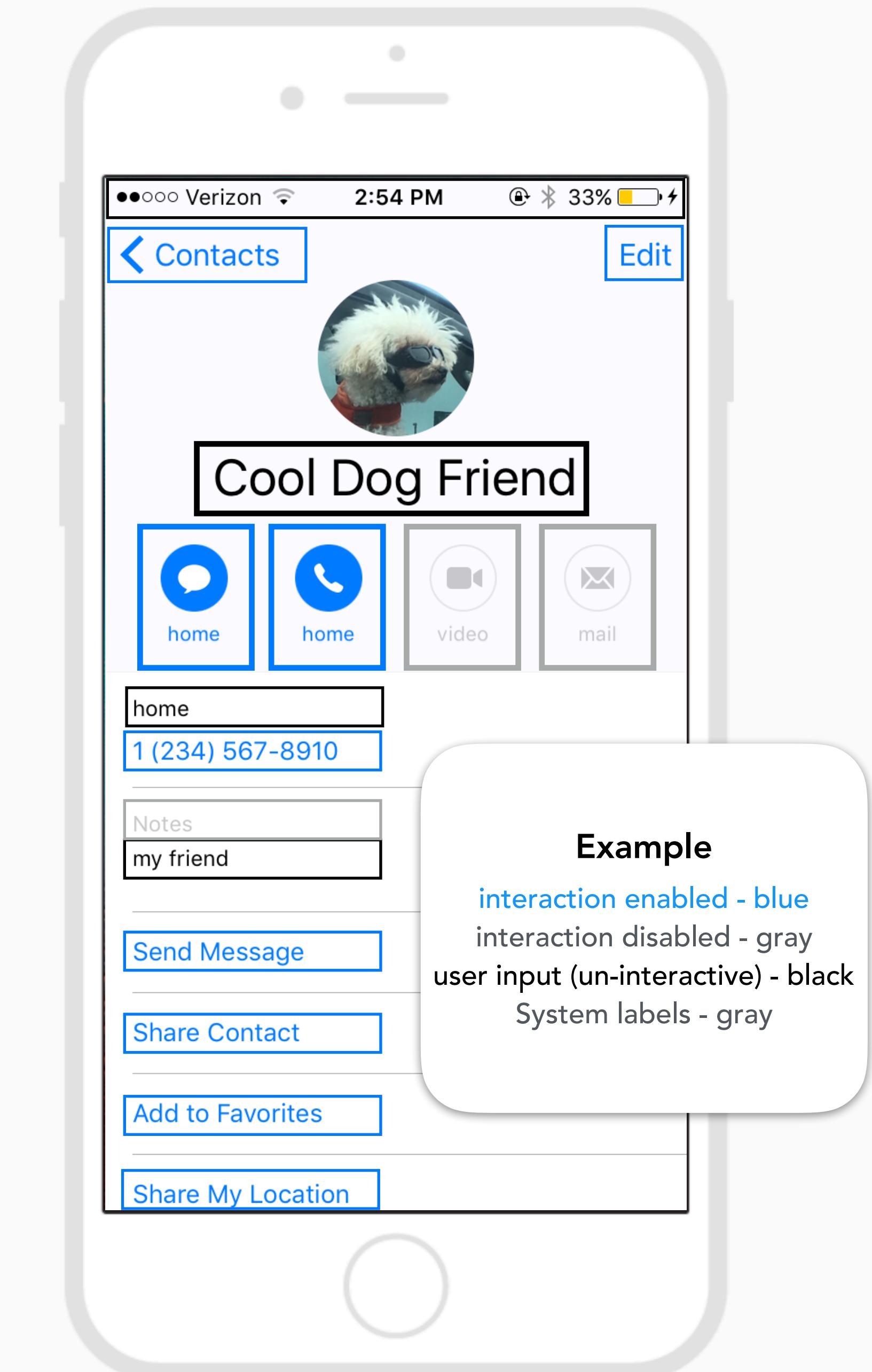


# iOS HIG : Color

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



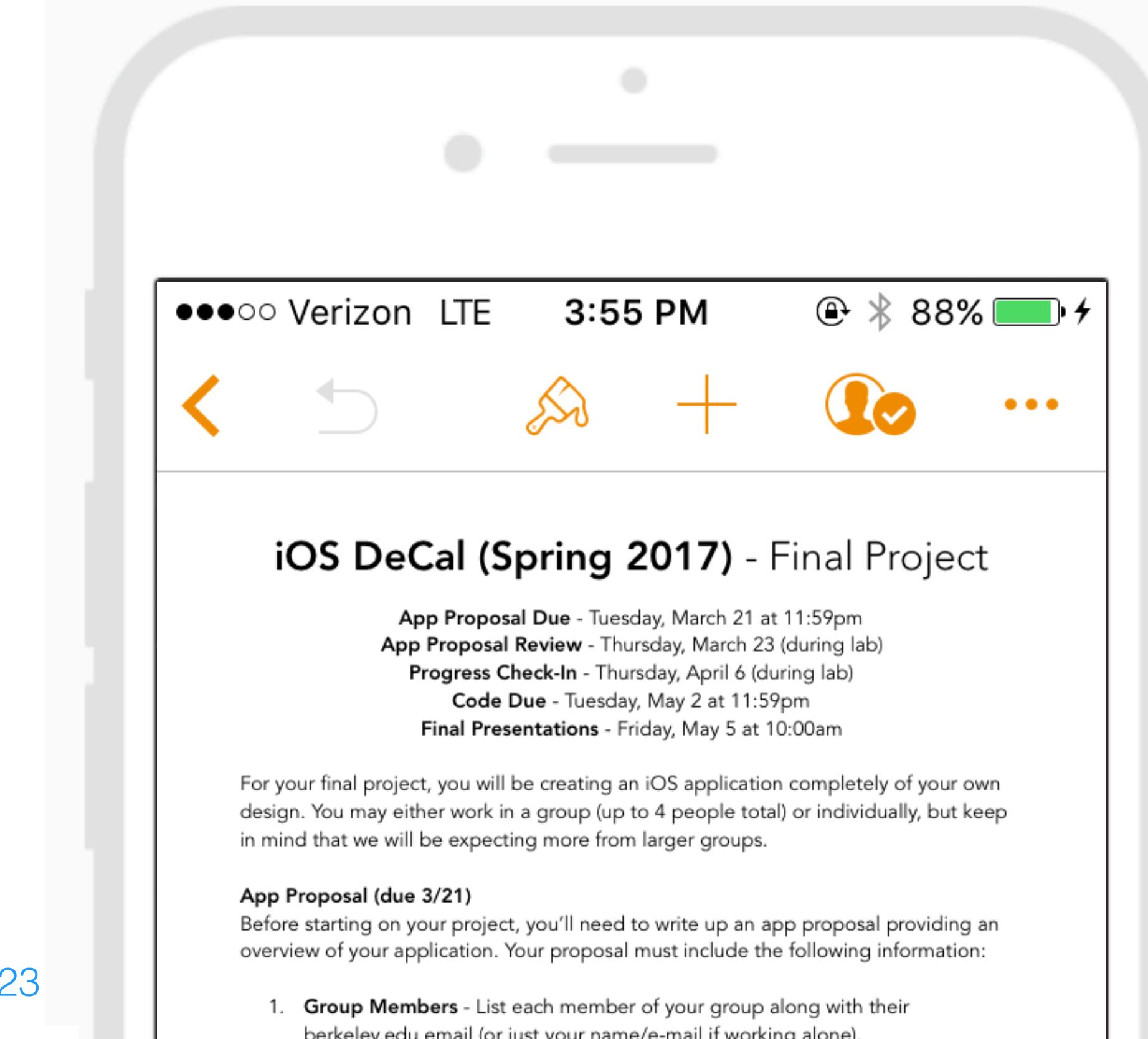
# iOS HIG : Color

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



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

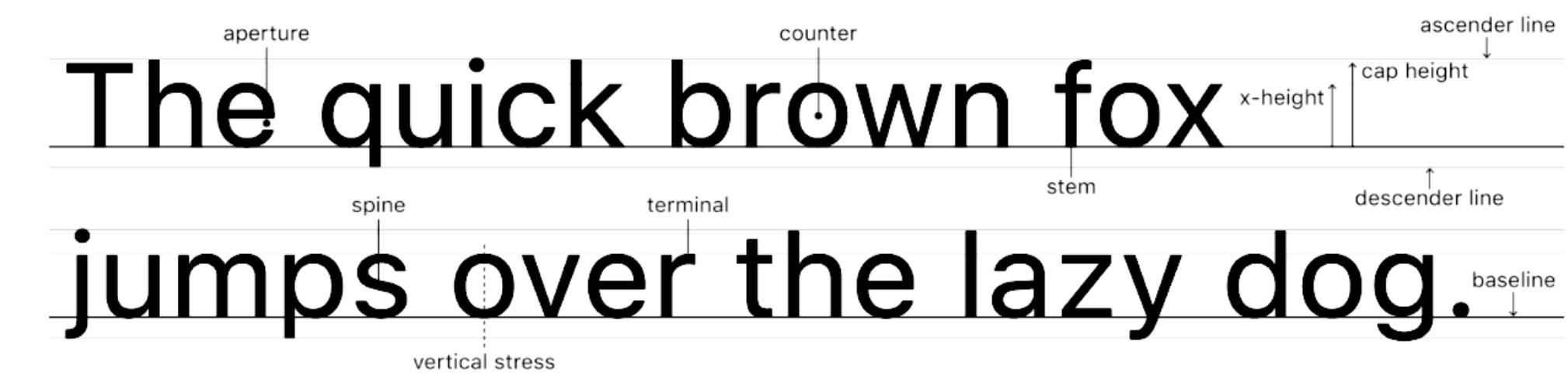
# iOS HIG : Color



Photoshop has accessibility color filters to help you do this

<http://www.adobe.com/accessibility/products/photoshop.html>

# iOS HIG : Fonts and Typography



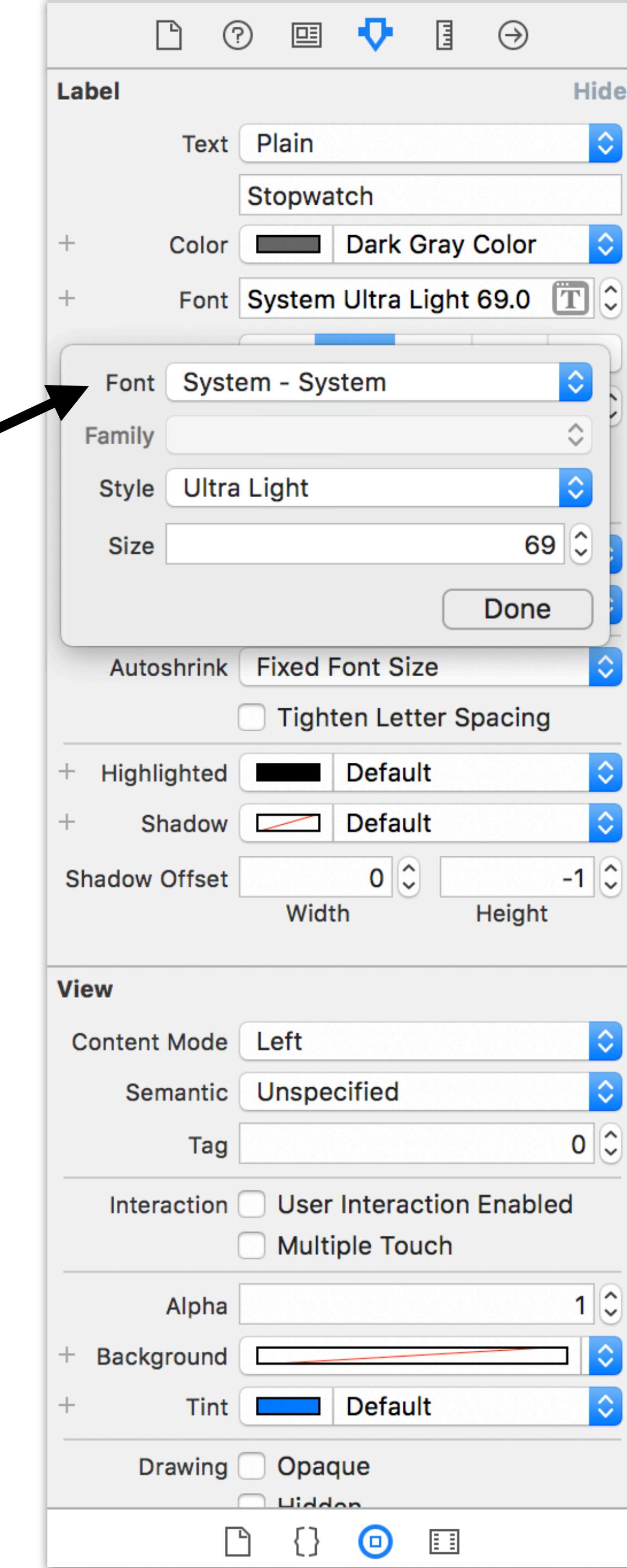
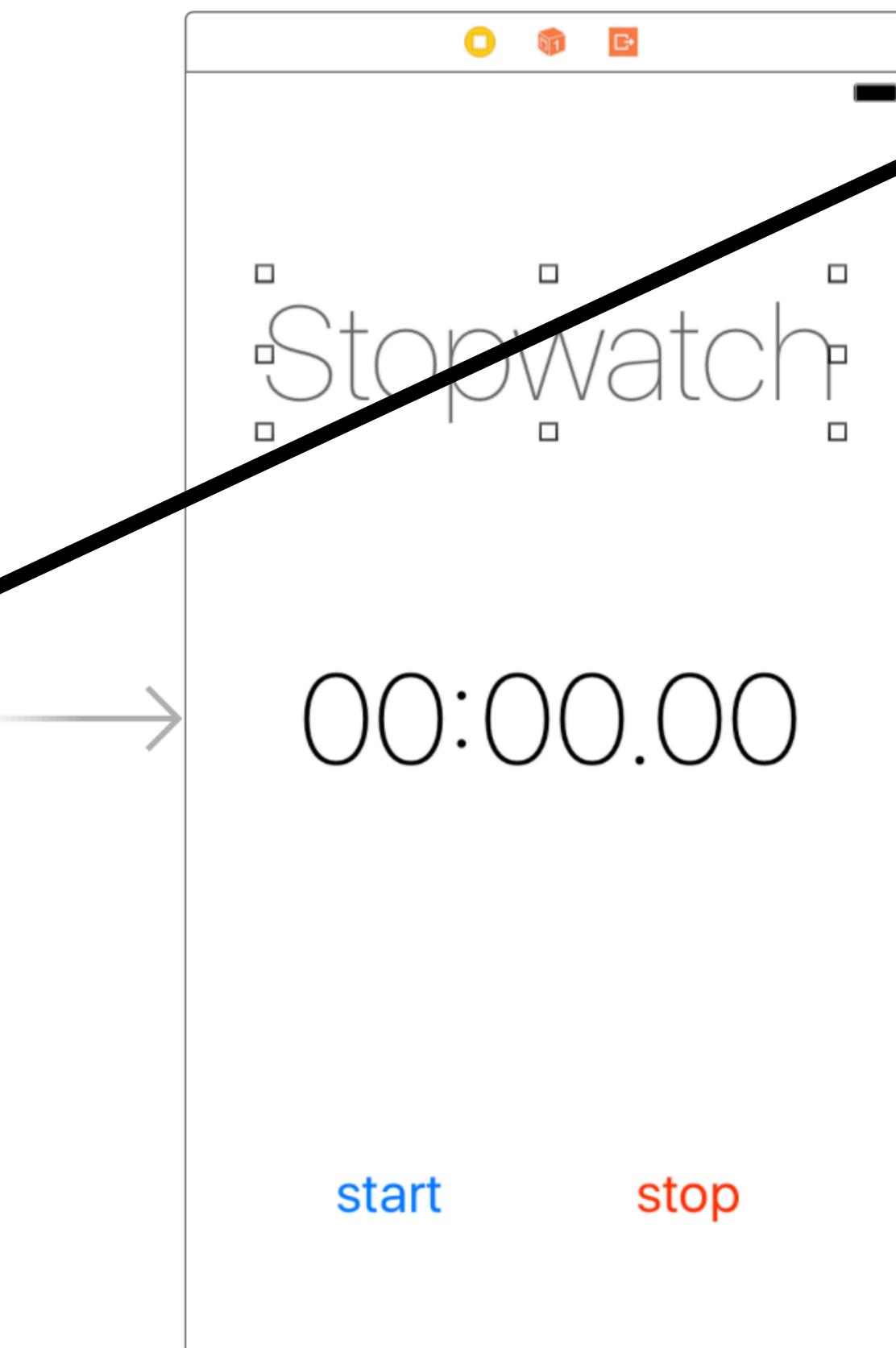
## San Francisco

The System Font for iOS

Created by a team at Apple in 2014

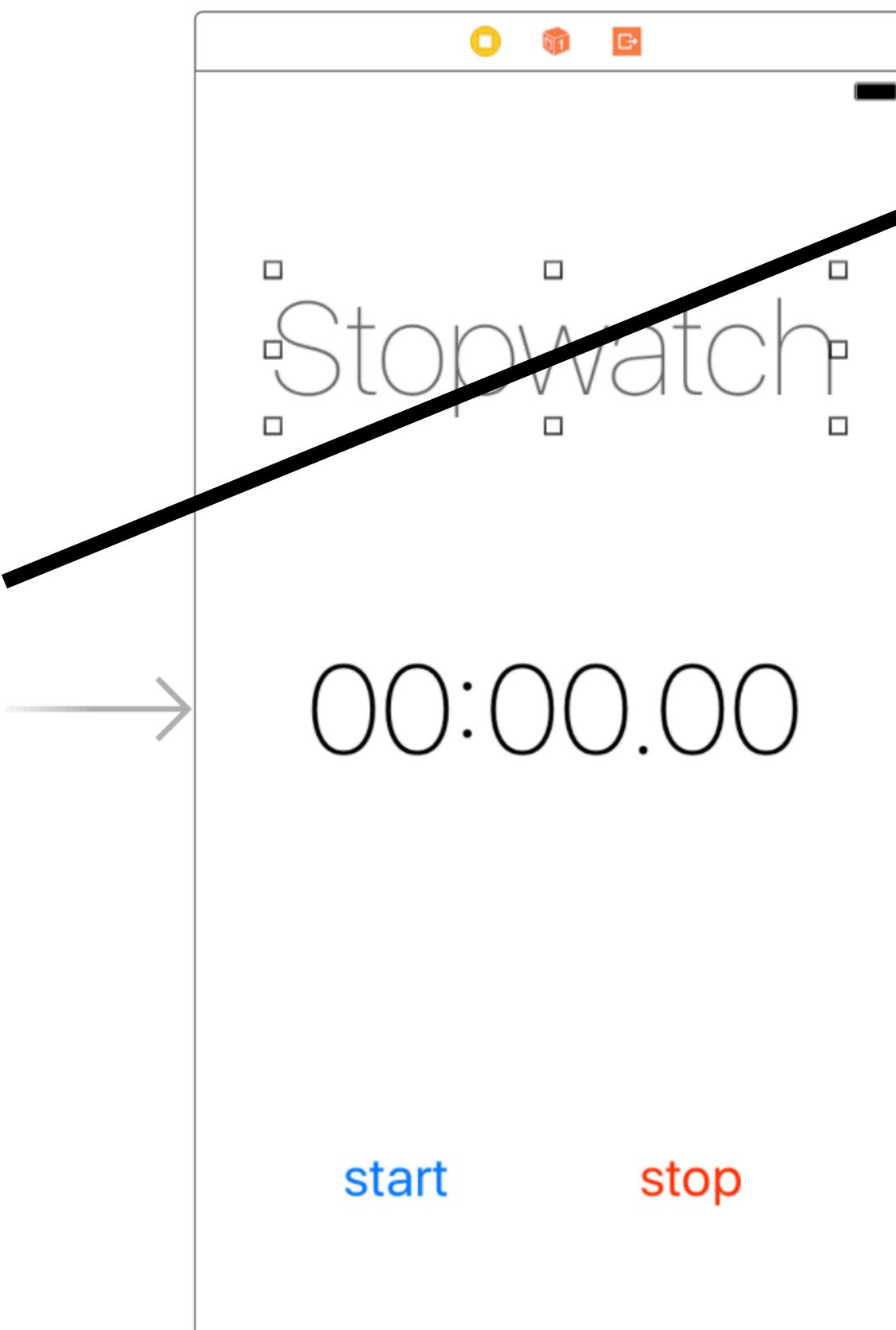
# iOS HIG : Fonts

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**



**Label**

Text Plain  
Stopwatch  
Color Dark Gray Color  
Font System Ultra Light 69.0

**Font** Custom  
Family Helvetica Neue  
Style Regular  
Size 17

Autoshrink Fixed Font Size  
Tighten Letter Spacing

Highlighted Default  
Shadow Default  
Shadow Offset 0 -1

**View**

Content Mode Left  
Semantic Unspecified  
Tag 0  
User Interaction Enabled  
Multiple Touch  
Alpha 1  
Background  
Tint Default  
Opaque  
Hidden

# 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**

# Check-In

# Programmatic Design

# **Storyboard : Review**

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

## **Pros of Storyboard**

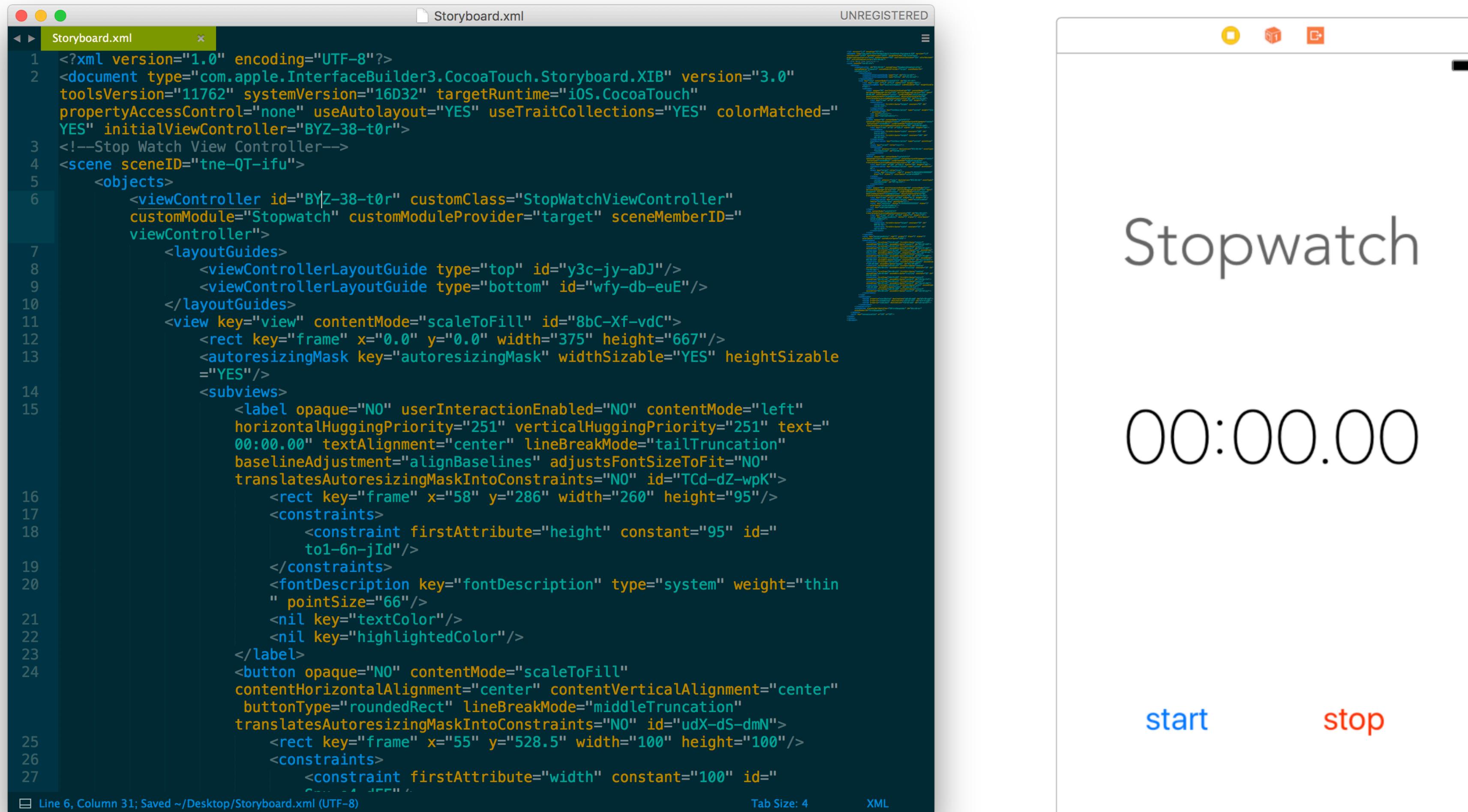
Drag and drop interface makes it really easy to visualize your application immediately

Relatively low learning curve

Great for small projects

The future of User Interface programming?

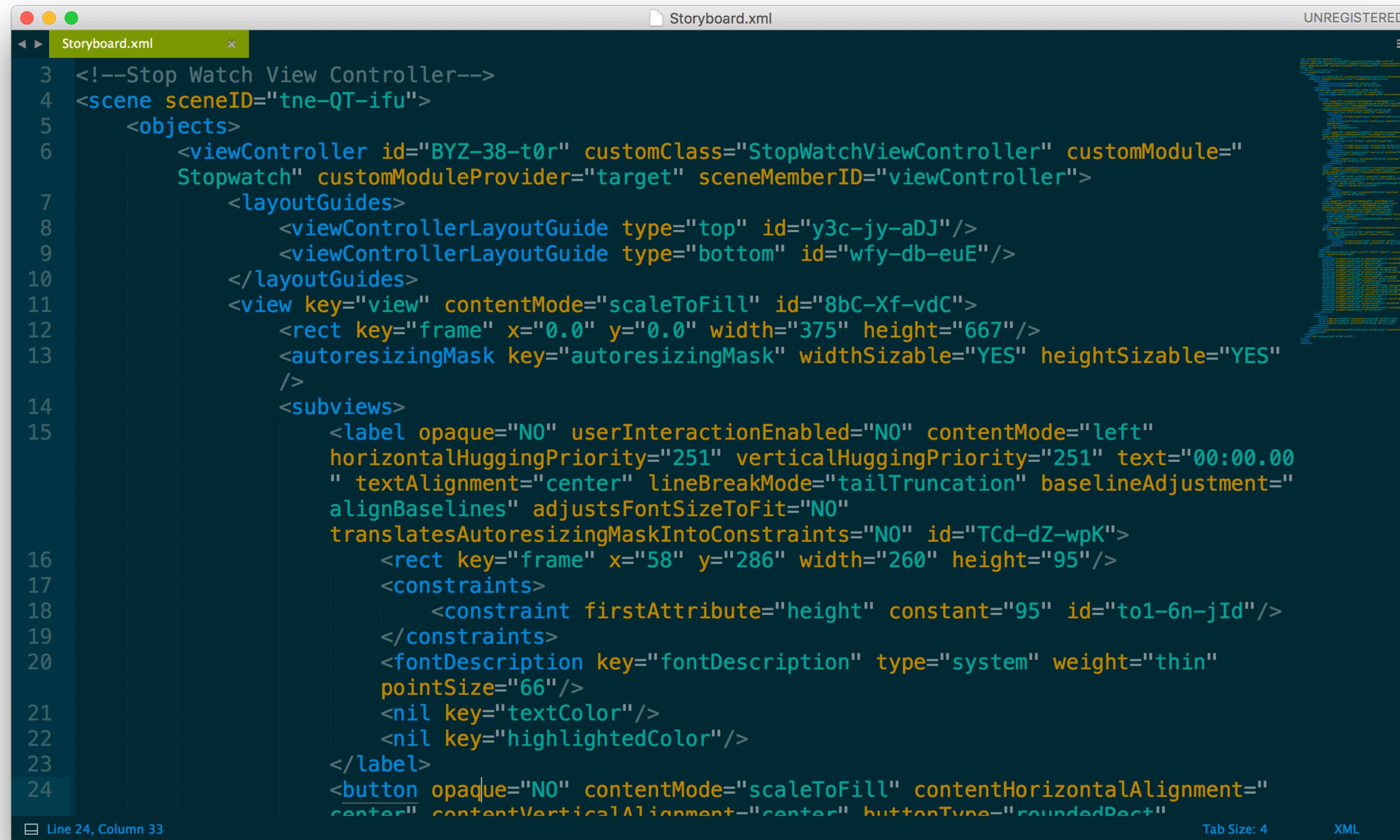
# Storyboard : Beneath the hood



The image shows a comparison between the XML representation of a storyboard and its visual preview. On the left, a screenshot of a Mac OS X desktop shows a text editor window titled "Storyboard.xml". The code is an XML document describing a storyboard scene. It includes elements like <viewController>, <layoutGuides>, <view>, <label>, and <button>. The XML uses standard tags and attributes to define the UI components and their properties. On the right, a screenshot of the Xcode storyboard preview interface shows a simple stopwatch screen with the title "Stopwatch" and a digital clock display showing "00:00.00". Below the display are two large buttons labeled "start" and "stop".

```
<?xml version="1.0" encoding="UTF-8"?>
<document type="com.apple.InterfaceBuilder3.CocoaTouch.Storyboard.XIB" version="3.0"
toolsVersion="11762" systemVersion="16D32" targetRuntime="iOS.CocoaTouch"
propertyAccessControl="none" useAutoLayout="YES" useTraitCollections="YES" colorMatched="YES"
initialViewController="BYZ-38-t0r">
<!--Stop Watch View Controller-->
<scene sceneID="tne-QT-ifu">
<objects>
<viewController id="BYZ-38-t0r" customClass="StopWatchViewController"
customModule="Stopwatch" customModuleProvider="target" sceneMemberID="viewController">
<layoutGuides>
<viewControllerLayoutGuide type="top" id="y3c-jy-aDJ"/>
<viewControllerLayoutGuide type="bottom" id="wfy-db-euE"/>
</layoutGuides>
<view key="view" contentMode="scaleToFill" id="8bC-Xf-vdC">
<rect key="frame" x="0.0" y="0.0" width="375" height="667"/>
<autoresizingMask key="autoresizingMask" widthSizable="YES" heightSizable="YES"/>
<subviews>
<label opaque="NO" userInteractionEnabled="NO" contentMode="left"
horizontalHuggingPriority="251" verticalHuggingPriority="251" text="00:00.00"
textAlignment="center" lineBreakMode="tailTruncation"
baselineAdjustment="alignBaselines" adjustsFontSizeToFit="NO"
translatesAutoresizingMaskIntoConstraints="NO" id="TCd-dZ-wpK">
<rect key="frame" x="58" y="286" width="260" height="95"/>
<constraints>
<constraint firstAttribute="height" constant="95" id="to1-6n-jId"/>
</constraints>
<fontDescription key="fontDescription" type="system" weight="thin"
pointSize="66"/>
<nil key="textColor"/>
<nil key="highlightedColor"/>
</label>
<button opaque="NO" contentMode="scaleToFill"
contentHorizontalAlignment="center" contentVerticalAlignment="center"
buttonType="roundedRect" lineBreakMode="middleTruncation"
translatesAutoresizingMaskIntoConstraints="NO" id="udX-dS-dmN">
<rect key="frame" x="55" y="528.5" width="100" height="100"/>
<constraints>
<constraint firstAttribute="width" constant="100" id="Gm-1A-dPf"/>
</constraints>
</button>
</subviews>
</view>
</viewController>
</objects>
</scene>
</document>
```

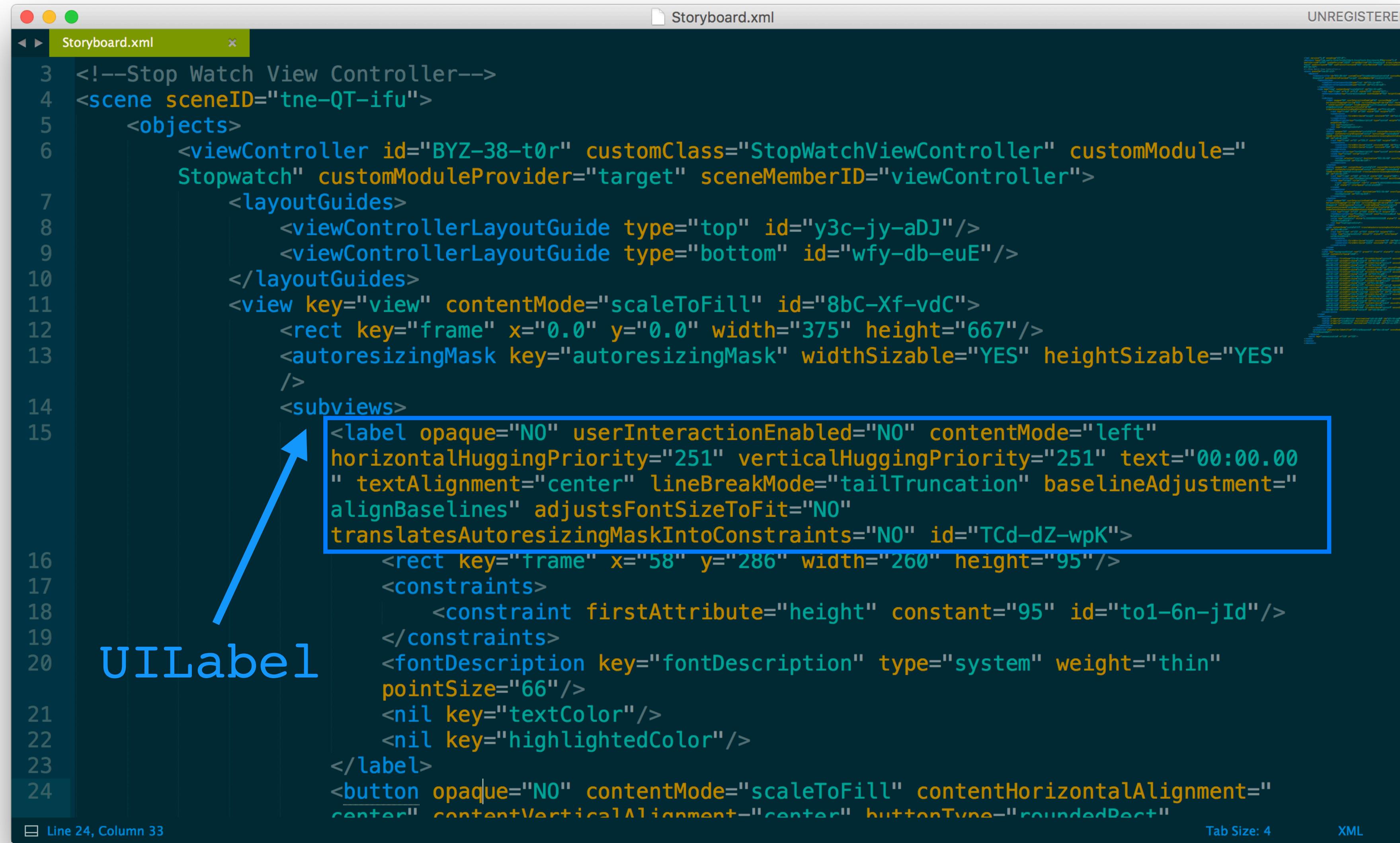
Main.storyboard files are just XML files  
35



The screenshot shows a text editor window with the title "Storyboard.xml". The code displayed is the XML representation of a storyboard scene. The scene contains a view controller with a specific ID and custom class, and a label and button subviews. The label has a text value of "00:00.00" and various styling properties like font size and color. The XML is color-coded for readability.

```
3 <!--Stop Watch View Controller-->
4 <scene sceneID="tne-QT-ifu">
5   <objects>
6     <viewController id="BYZ-38-t0r" customClass="StopWatchViewController" customModule="Stopwatch" customModuleProvider="target" sceneMemberID="viewController">
7       <layoutGuides>
8         <viewControllerLayoutGuide type="top" id="y3c-jy-aDJ"/>
9         <viewControllerLayoutGuide type="bottom" id="wfy-db-euE"/>
10      </layoutGuides>
11      <view key="view" contentMode="scaleToFill" id="8bC-Xf-vdC">
12        <rect key="frame" x="0.0" y="0.0" width="375" height="667"/>
13        <autoresizingMask key="autoresizingMask" widthSizable="YES" heightSizable="YES" />
14      <subviews>
15        <label opaque="NO" userInteractionEnabled="NO" contentMode="left" horizontalHuggingPriority="251" verticalHuggingPriority="251" text="00:00.00" textAlignment="center" lineBreakMode="tailTruncation" baselineAdjustment="alignBaselines" adjustsFontSizeToFit="NO" translatesAutoresizingMaskIntoConstraints="NO" id="TCd-dZ-wpK">
16          <rect key="frame" x="58" y="286" width="260" height="95"/>
17          <constraints>
18            <constraint firstAttribute="height" constant="95" id="to1-6n-jId"/>
19          </constraints>
20          <fontDescription key="fontDescription" type="system" weight="thin" pointSize="66"/>
21          <nil key="textColor"/>
22          <nil key="highlightedColor"/>
23        </label>
24        <button opaque="NO" contentMode="scaleToFill" contentHorizontalAlignment="center" contentVerticalAlignment="center" buttonType="roundedRect" contentHorizontalAlignment="center" contentVerticalAlignment="center" id="iLJ-9P-9DQ">
<!--End Stop Watch View Controller-->
```

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



```
Storyboard.xml UNREGISTERED  
3 <!--Stop Watch View Controller-->  
4 <scene sceneID="tne-QT-ifu">  
5   <objects>  
6     <viewController id="BYZ-38-t0r" customClass="StopWatchViewController" customModule="Stopwatch" customModuleProvider="target" sceneMemberID="viewController">  
7       <layoutGuides>  
8         <viewControllerLayoutGuide type="top" id="y3c-jy-aDJ"/>  
9         <viewControllerLayoutGuide type="bottom" id="wfy-db-euE"/>  
10    </layoutGuides>  
11    <view key="view" contentMode="scaleToFill" id="8bC-Xf-vdC">  
12      <rect key="frame" x="0.0" y="0.0" width="375" height="667"/>  
13      <autoresizingMask key="autoresizingMask" widthSizable="YES" heightSizable="YES" />  
14    <subviews>  
15      <label opaque="NO" userInteractionEnabled="NO" contentMode="left" horizontalHuggingPriority="251" verticalHuggingPriority="251" text="00:00.00" textAlignment="center" lineBreakMode="tailTruncation" baselineAdjustment="alignBaselines" adjustsFontSizeToFit="NO" translatesAutoresizingMaskIntoConstraints="NO" id="TCd-dZ-wpK">  
16        <rect key="frame" x="58" y="286" width="260" height="95"/>  
17        <constraints>  
18          <constraint firstAttribute="height" constant="95" id="to1-6n-jId"/>  
19        </constraints>  
20        <fontDescription key="fontDescription" type="system" weight="thin" pointSize="66"/>  
21        <nil key="textColor"/>  
22        <nil key="highlightedColor"/>  
23      </label>  
24      <button opaque="NO" contentMode="scaleToFill" contentHorizontalAlignment="center" contentVerticalAlignment="center" buttonType="roundedRect" contentHorizontalAlignment="center" contentVerticalAlignment="center" id="tPj-9g-9Dw">  
Line 24, Column 33 Tab Size: 4 XML
```

UILabel

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



The screenshot shows a GitHub Gist page for user 'neilinglis' with the commit hash 'gist:e238d5f22f85fa259ade'. It was created 3 years ago. There are two tabs: 'Code' (selected) and 'Revisions 1'. A note at the top says 'Storyboard Merge Conflict. Is there any sensible course of action for this?'. The code in 'gistfile1.txt' is as follows:

```
1 <<<<< HEAD
2     <segue reference="kXa-Mw-CAj"/>
3     <segue reference="TDo-1S-nUS"/>
4     <segue reference="hJU-8t-Kde"/>
5     <segue reference="haI-hu-Unh"/>
6     <segue reference="2ra-9a-Rv0"/>
7     <segue reference="ixW-dA-JnA"/>
8 =====
9     <segue reference="BwM-Nh-uZ9"/>
10    <segue reference="YWK-Ch-1fU"/>
11    <segue reference="haI-hu-Unh"/>
12    <segue reference="TDo-1S-nUS"/>
13    <segue reference="hJU-8t-Kde"/>
14    <segue reference="y7Z-qu-r0P"/>
15 >>>> e9a57872e96f17a8d2d785e4de0132e75229a262
```

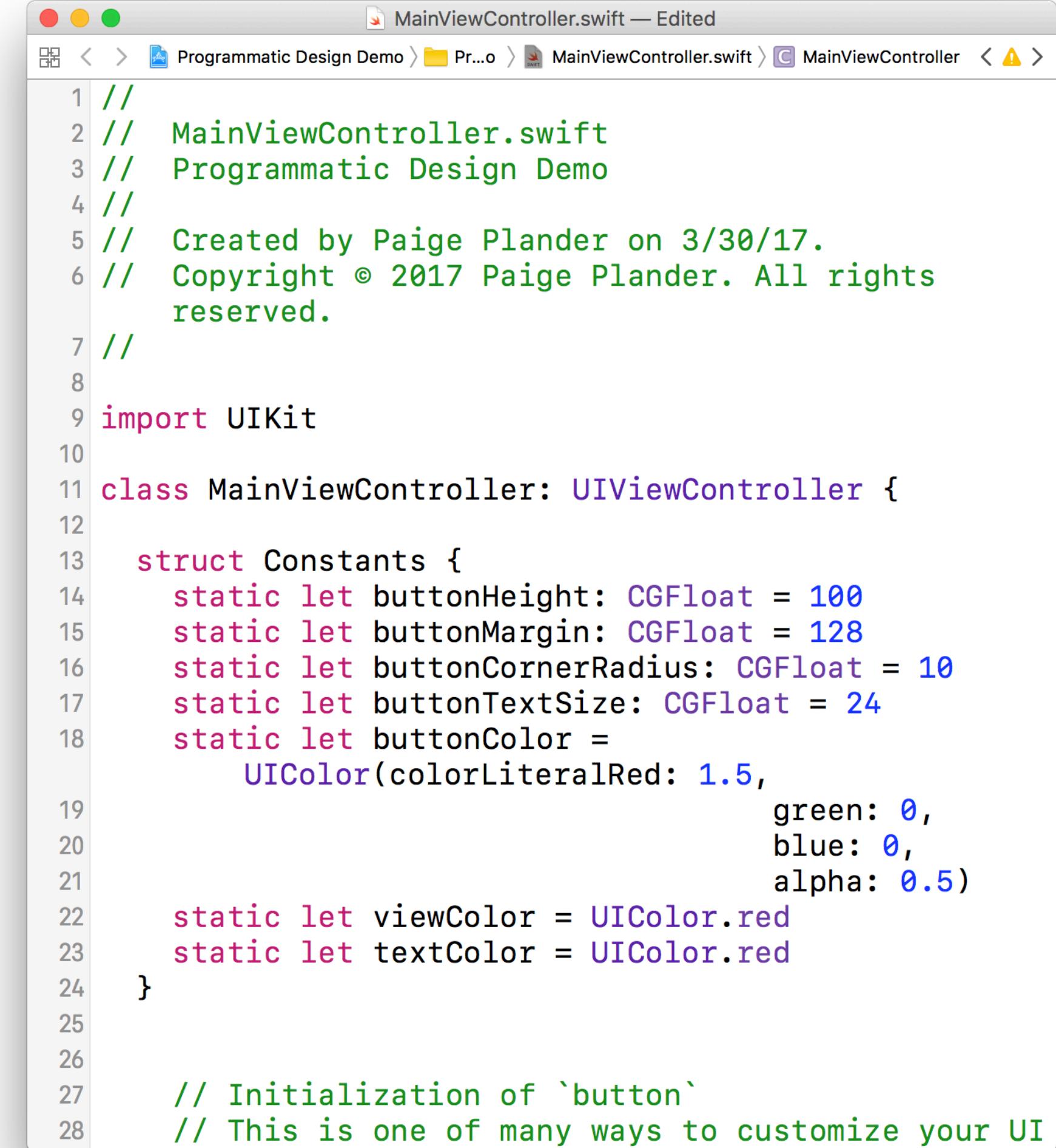
# Storyboard : Problems

## 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



The screenshot shows a Xcode editor window with the file 'MainViewController.swift' open. The code defines a class 'MainViewController' that inherits from 'UIViewController'. It includes a 'Constants' struct with static let properties for button height, margin, corner radius, text size, and colors. The code also initializes a 'button' with these constants.

```
// MainViewController.swift
// Programmatic Design Demo
//
// Created by Paige Plander on 3/30/17.
// Copyright © 2017 Paige Plander. All rights reserved.

import UIKit

class MainViewController: UIViewController {

    struct Constants {
        static let buttonHeight: CGFloat = 100
        static let buttonMargin: CGFloat = 128
        static let buttonCornerRadius: CGFloat = 10
        static let buttonTextSize: CGFloat = 24
        static let buttonColor =
            UIColor(colorLiteralRed: 1.5,
                    green: 0,
                    blue: 0,
                    alpha: 0.5)
        static let viewColor = UIColor.red
        static let textColor = UIColor.red
    }

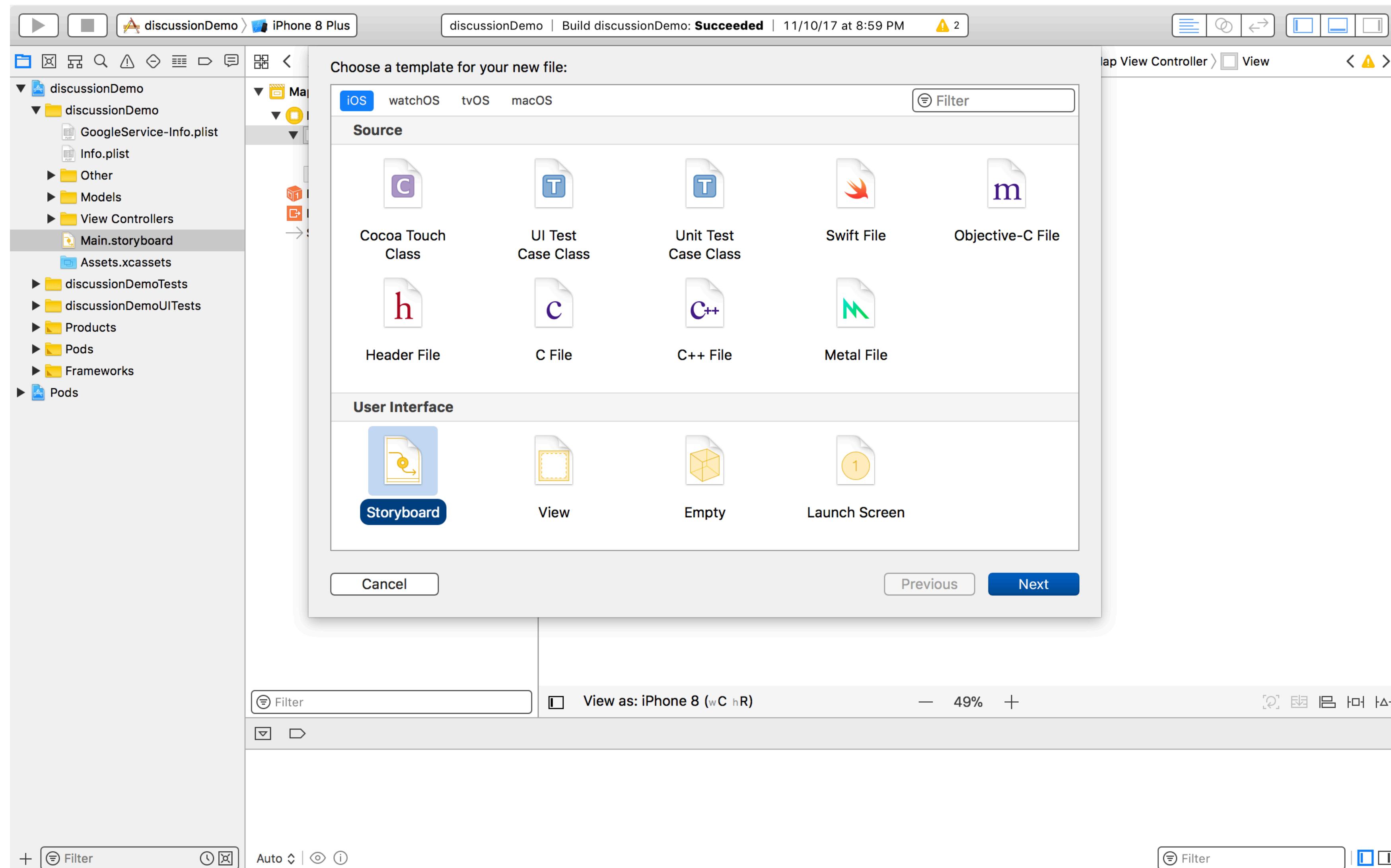
    // Initialization of `button`
    // This is one of many ways to customize your UI
}
```

# Storyboards References

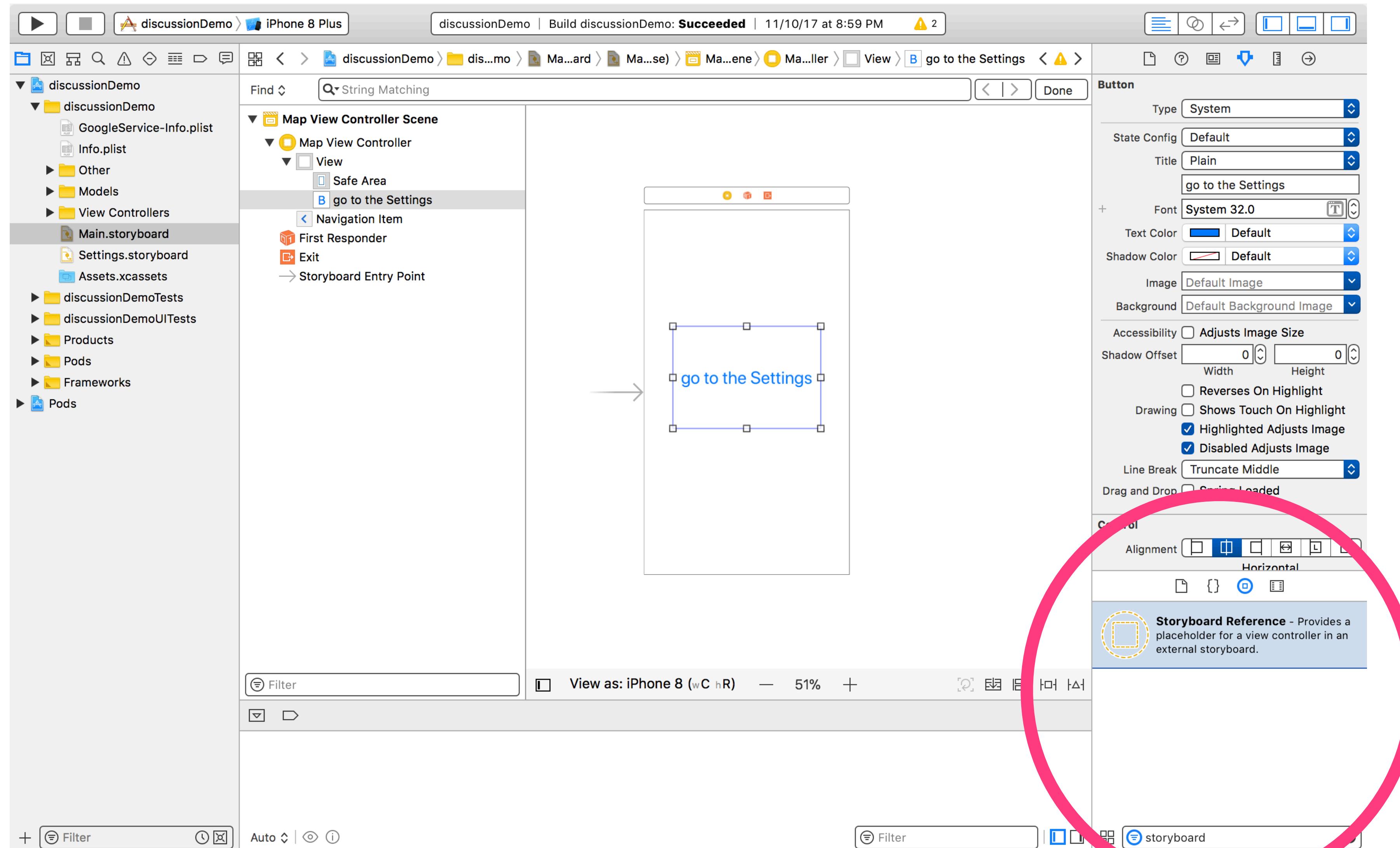
If you have a large project, break it up into several different Storyboard files.

Use “Storyboard References” to transition between Storyboards

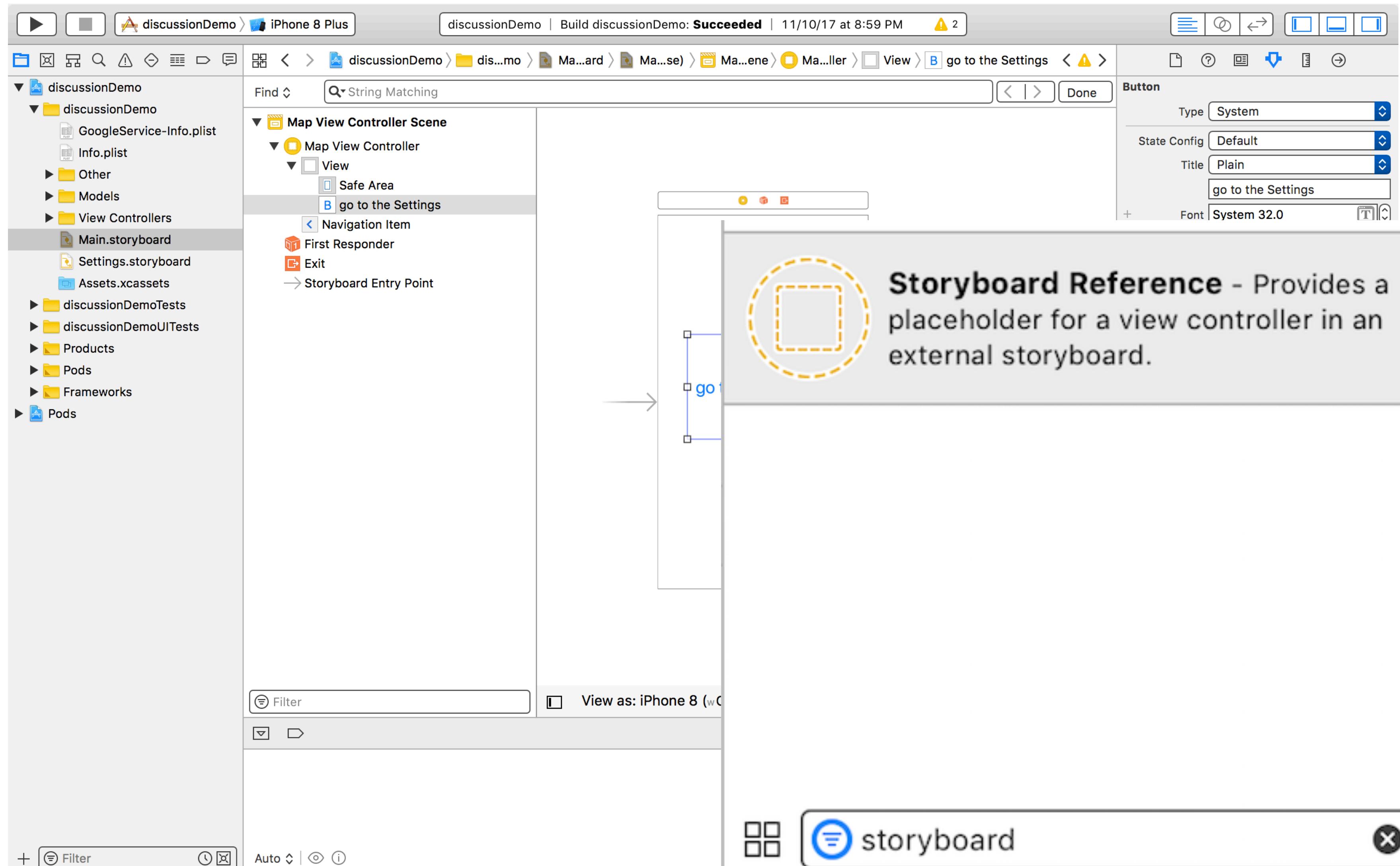
# Create the storyboard file



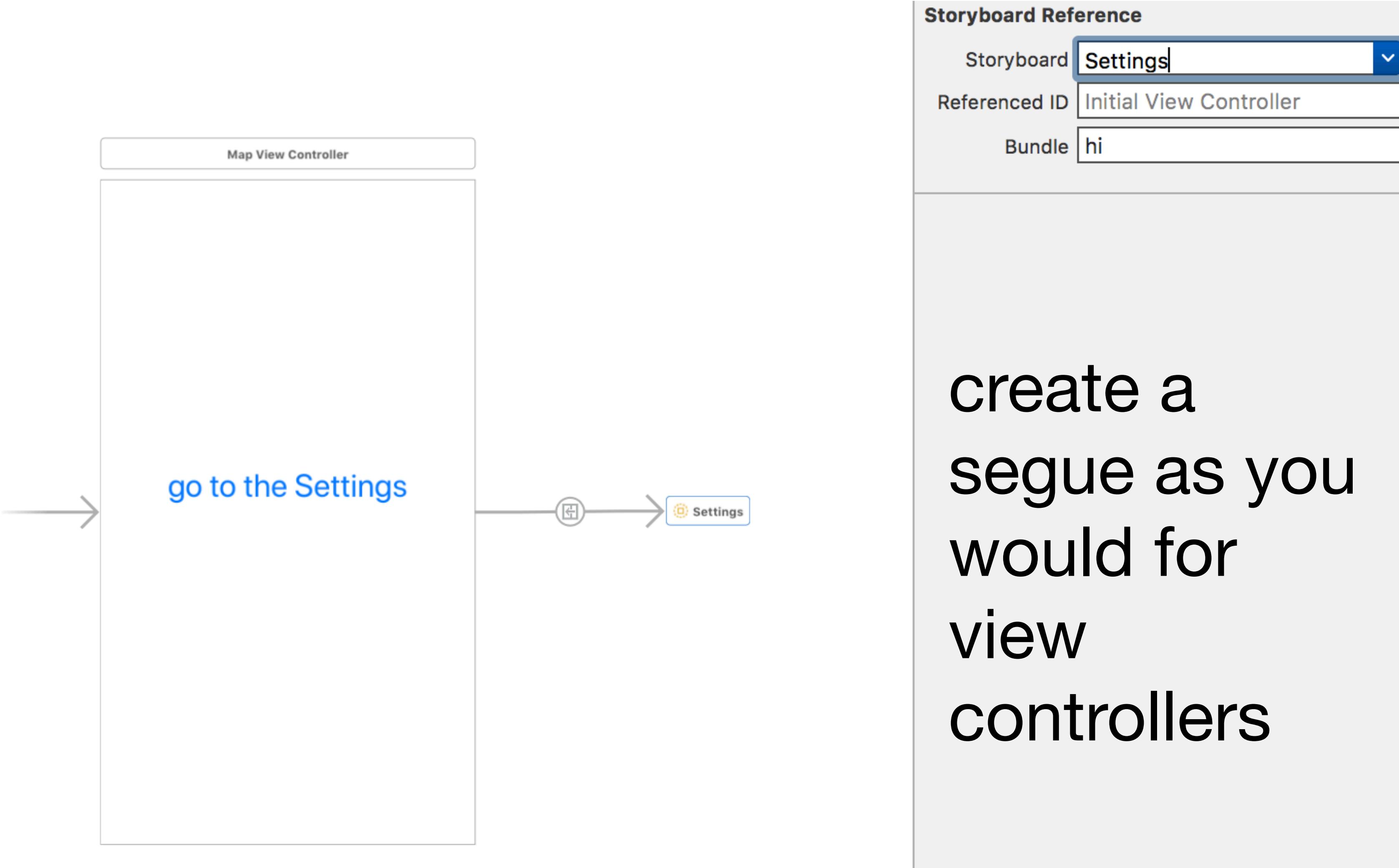
# Create a Storyboard Reference



# Create a Storyboard Reference



# Create a Storyboard Reference



# 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)

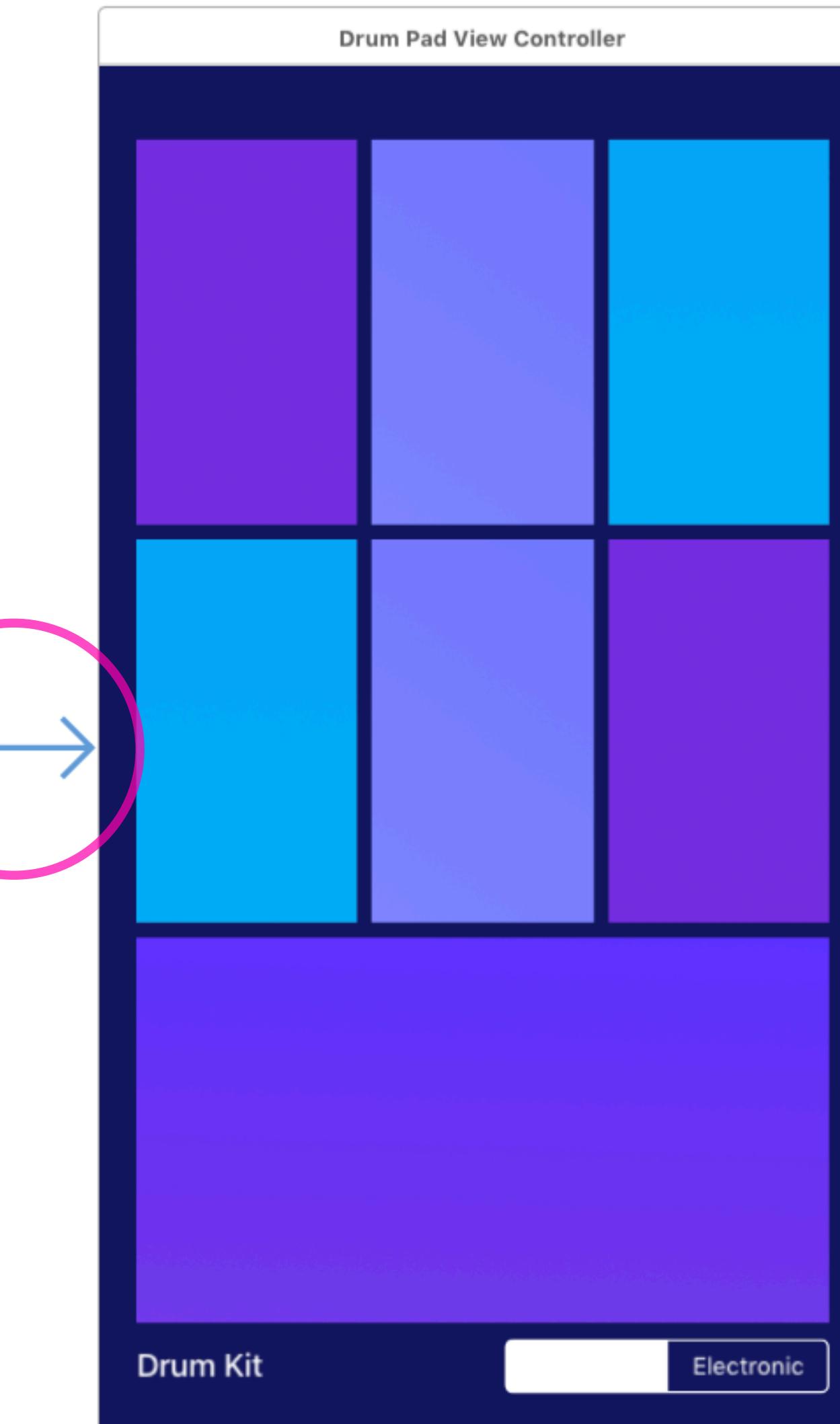
**UIScreen** - defines the properties of the user's device (get the bounds of user's device using `UIScreen.main.bounds`)

**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 its 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)

```
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  
    }  
    ...
```

Found in AppDelegate.swift

Setting your initial View Controller  
Programmatically (Example)

# Programmatic Design (with no Storyboard)

```
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  
    }  
    ...
```

Found in AppDelegate.swift

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

# Programmatic Design (with no Storyboard)

```
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  
    }  
    ...
```

Found in AppDelegate.swift

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

# Programmatic Design (with no Storyboard)

```
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  
    }  
    ...
```

Found in AppDelegate.swift

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

# Programmatic Design (with no Storyboard)

```
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  
    }  
    ...
```

Found in AppDelegate.swift

Set the window's  
root view controller property

# Programmatic Design (with no Storyboard)

```
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  
    }  
    ...
```

Found in AppDelegate.swift

Make the window visible to the user

# Programmatic Design (with no Storyboard)

```
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  
    }  
    ...
```

Found in AppDelegate.swift

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

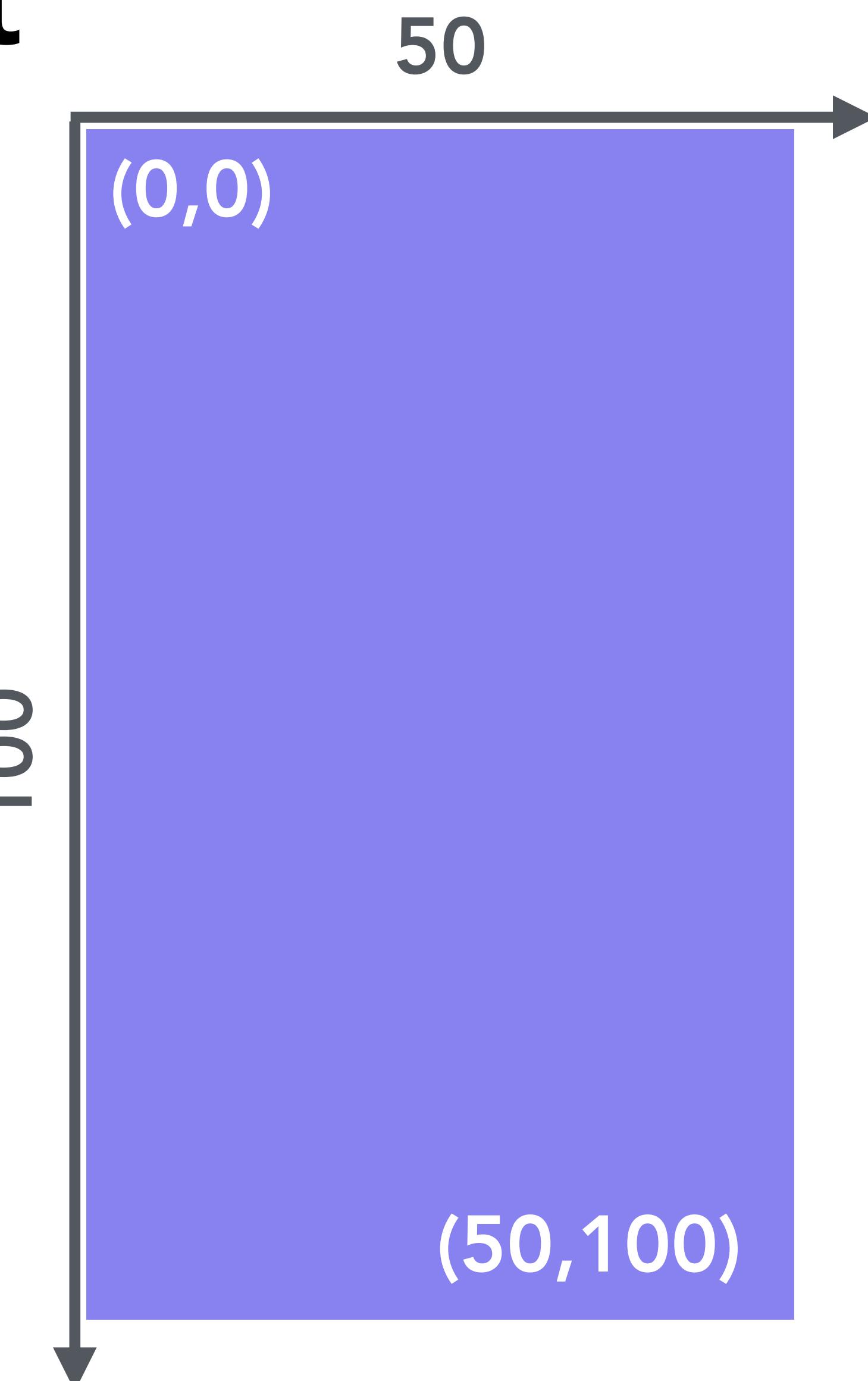
```
CGRect(x: 0, y: 0,  
       width: 100,  
       height: 100)
```

CGPoint - defines a position

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

CGSize - defines a size

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



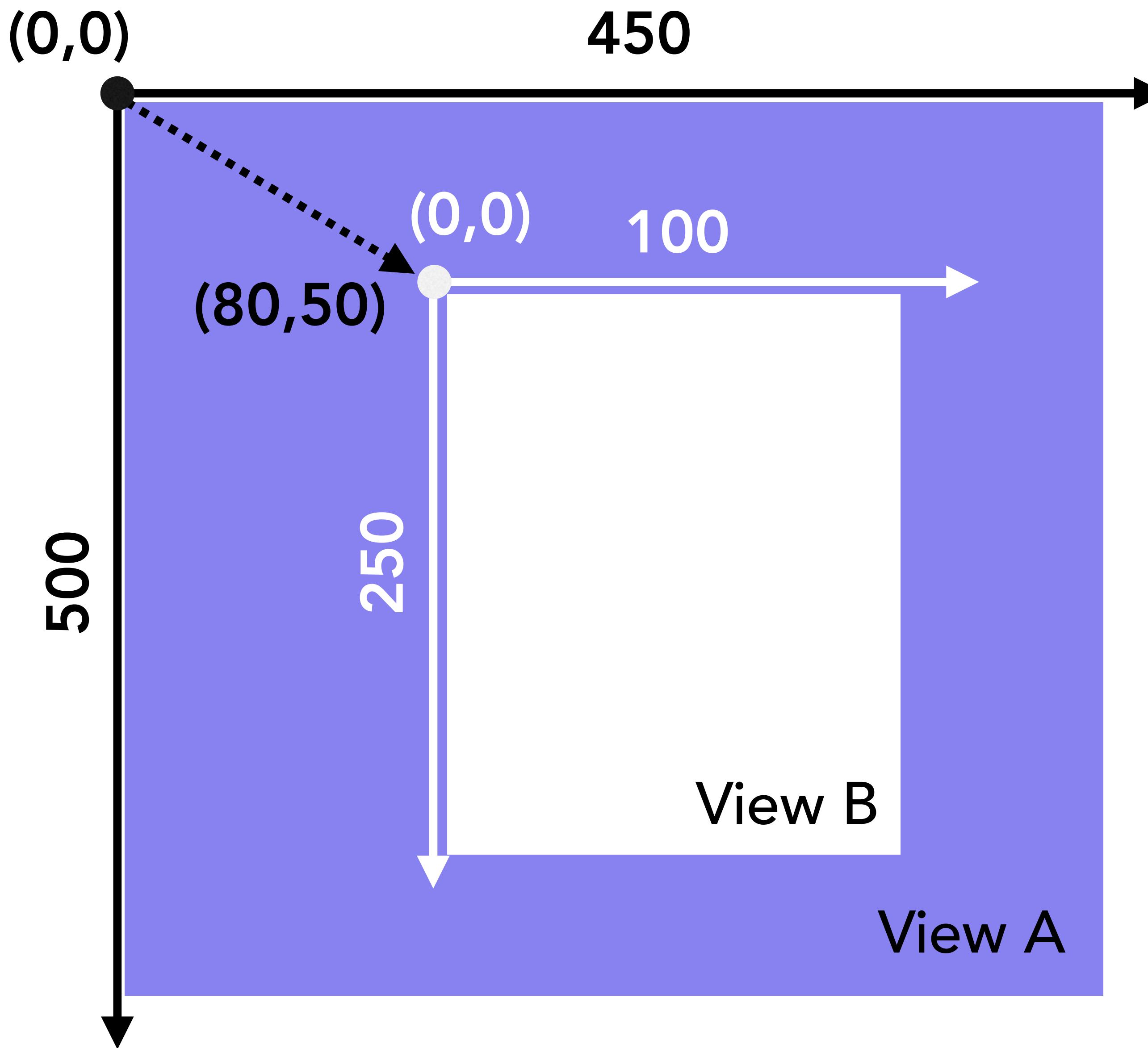
# UIView : Geometry

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

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

**bounds**: `CGRect` - 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**

$x, y = (0,0)$

width = 450

height = 500

**View A bounds**

$x, y = (0,0)$

width = 450

height = 500

**View B frame**

$x, y = (80,50)$

width = 100

height = 250

**View B bounds**

$x, y = (0,0)$

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)
```

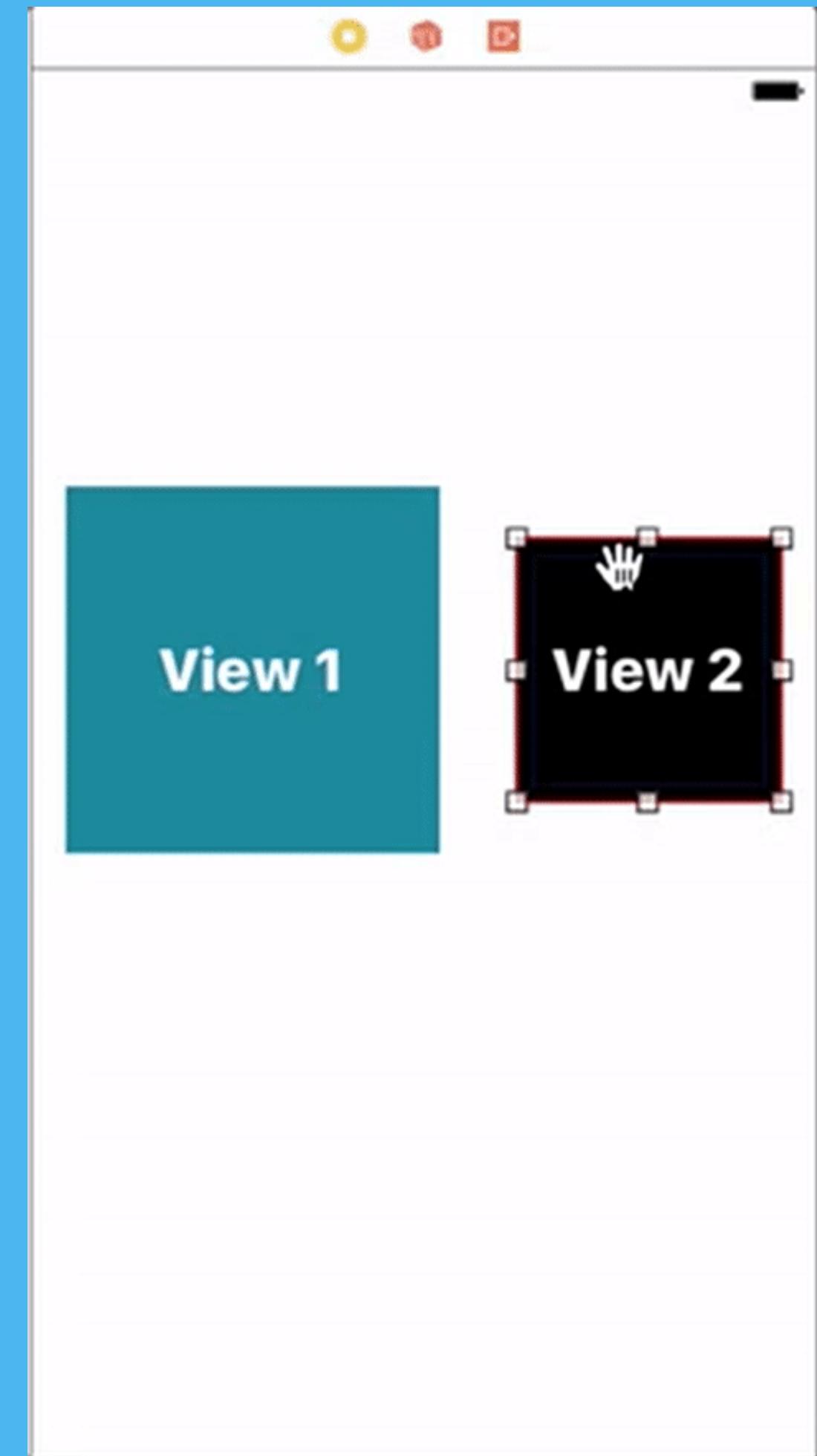
In this example, we create a list of constraints, then batch activate them (rather than doing it one by one)

# Programmatic AutoLayout

## Layout Anchors

```
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

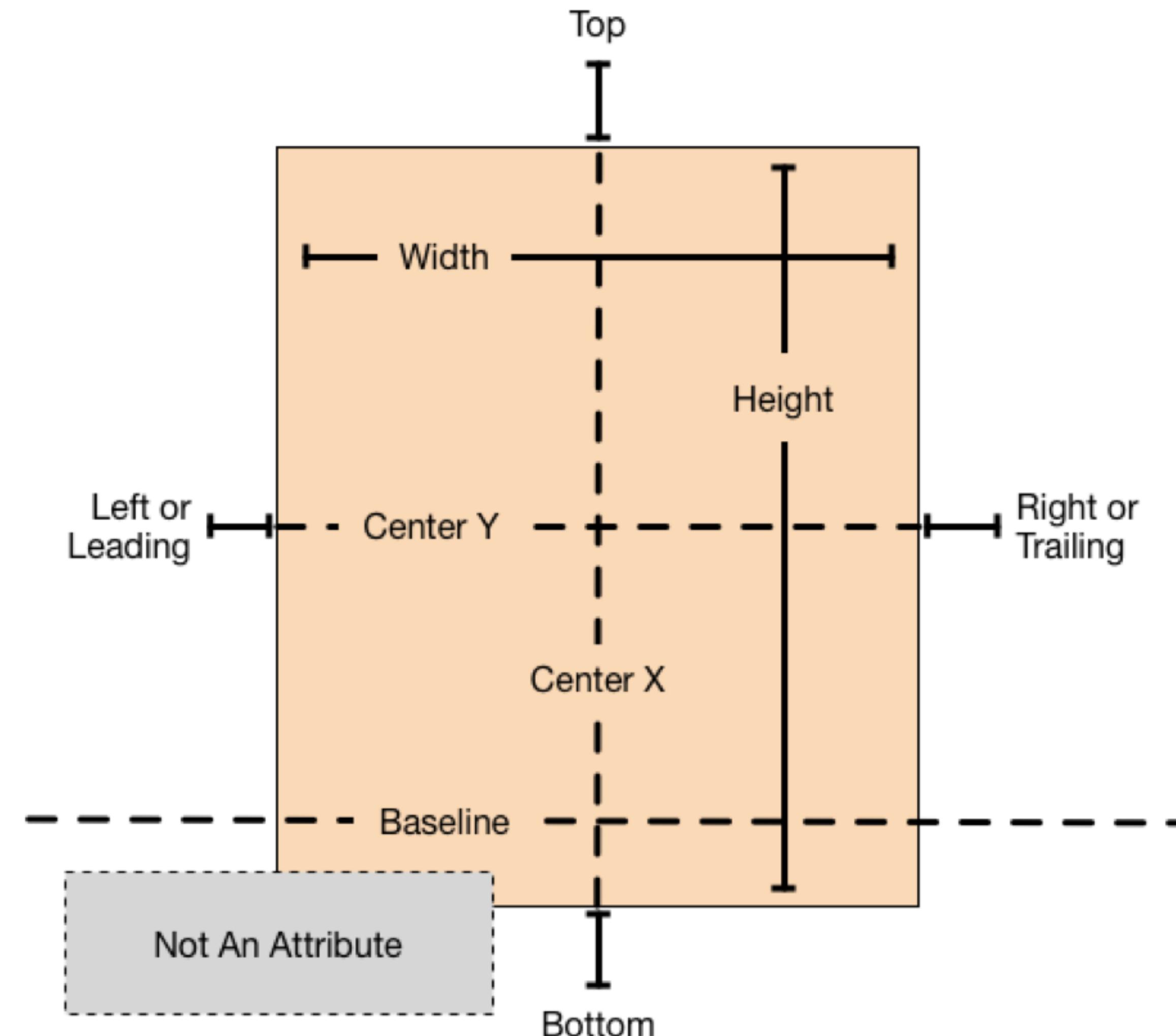


Equivalent Storyboard  
Example

# 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 your 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(prevVC, animated: true)
```

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

# Demo