

iOS102

# Getting started with the iOS Designer

- ▶ Lecture will begin shortly
- ▶ Download class materials from [university.xamarin.com](http://university.xamarin.com)



Information in this document is subject to change without notice. The example companies, organizations, products, people, and events depicted herein are fictitious. No association with any real company, organization, product, person or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user.

Xamarin may have patents, patent applications, trademarked, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any license agreement from Xamarin, the furnishing of this document does not give you any license to these patents, trademarks, or other intellectual property.

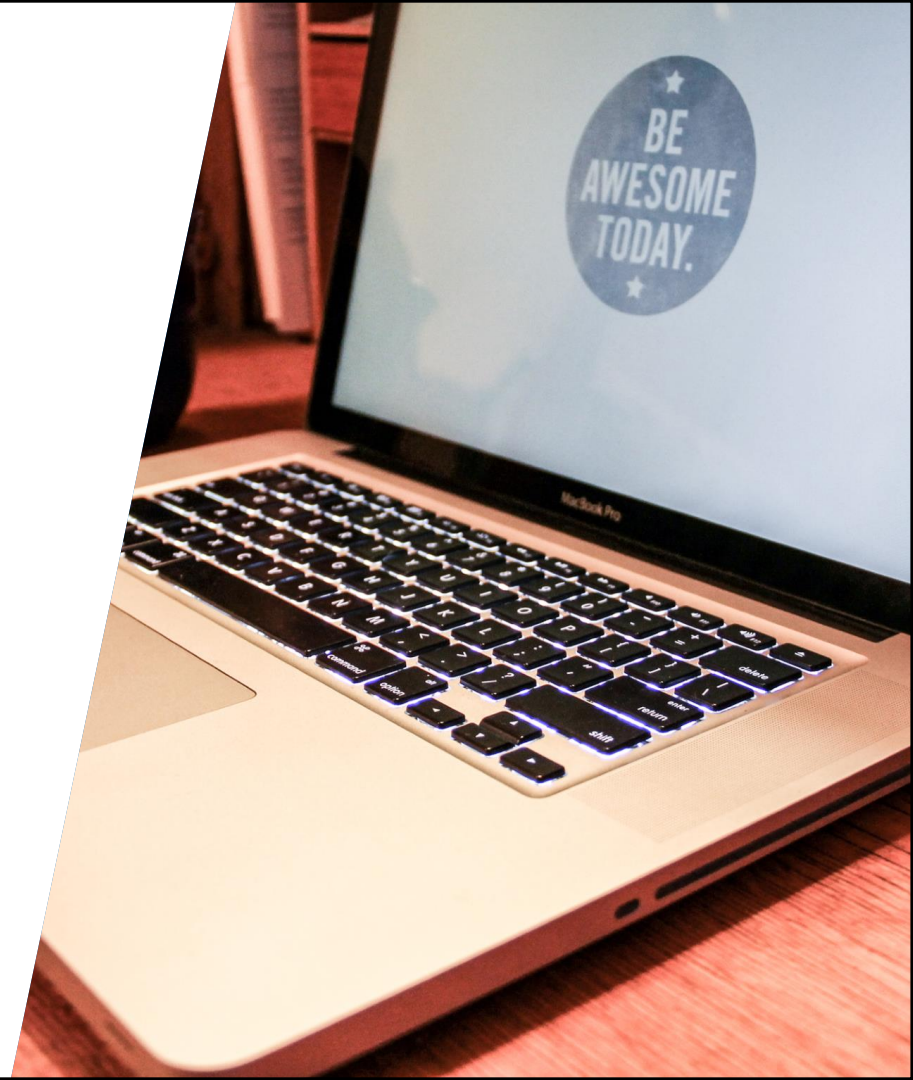
© 2015 Xamarin. All rights reserved.

Xamarin, MonoTouch, MonoDroid, Xamarin.iOS, Xamarin.Android, and Xamarin Studio are either registered trademarks or trademarks of Xamarin in the U.S.A. and/or other countries.

Other product and company names herein may be the trademarks of their respective owners.

# Objectives

1. Create a single screen application and add controls
2. Describe and use Auto Layout
3. Interact with controls and views programmatically
4. Apply segues and navigation





Create a single screen  
application and add controls



**Xamarin**  
University

# Tasks

1. Describe the iOS Designer
2. Identify controls and properties
3. Demonstrate the designer workflow
4. Work with subviews

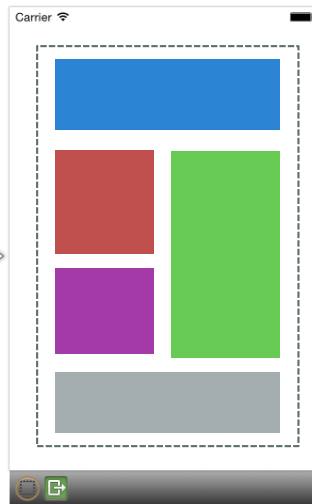


# Reminder: UIView

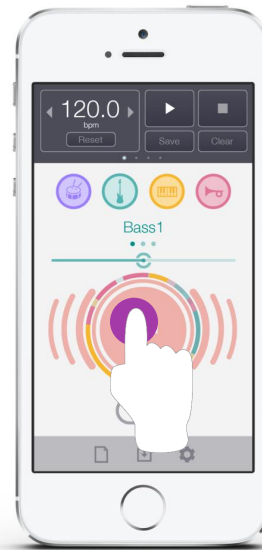
- ❖ A **UIView** defines a rectangular area on the screen and provides:



Visualization



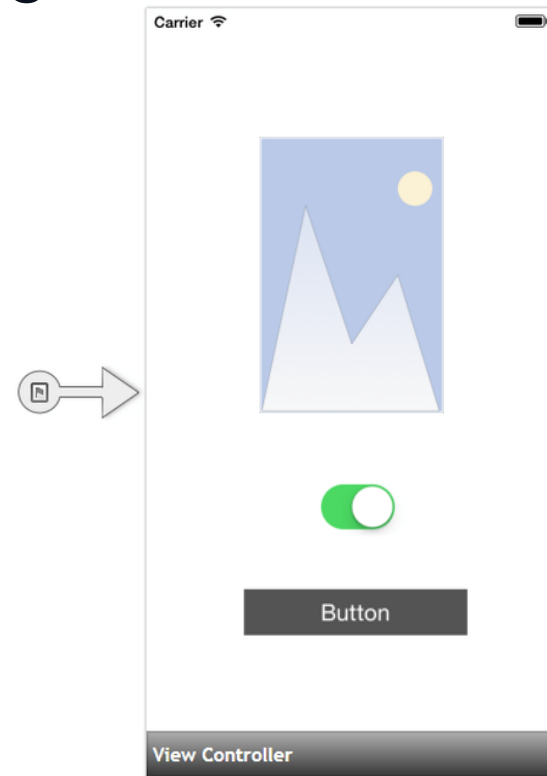
Layout for subviews



Event publishing

# Reminder: View Controllers

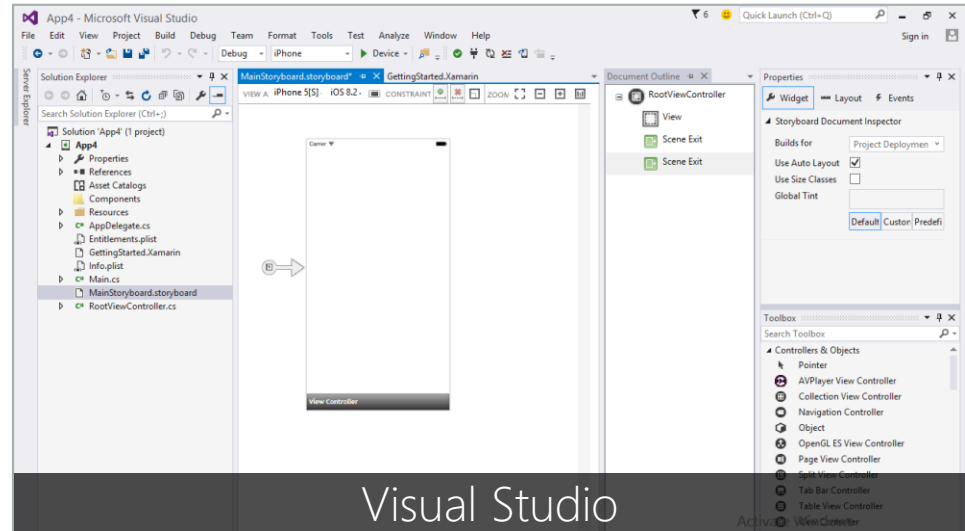
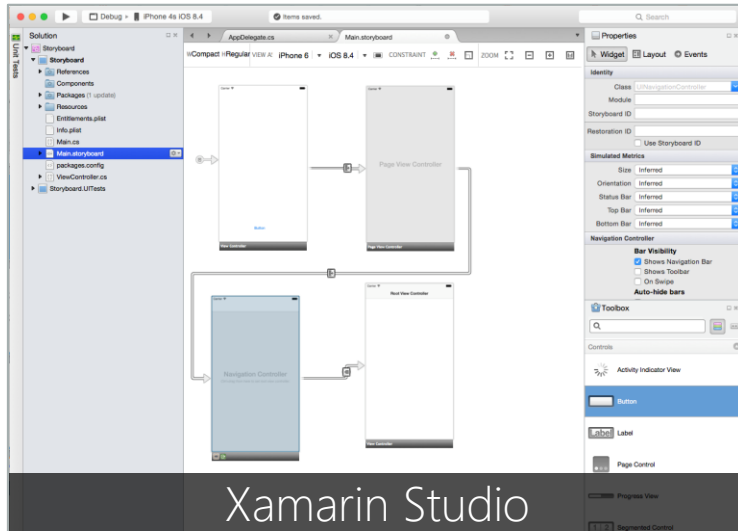
- ❖ A **UIViewController** provides view management for a single screen
- ❖ Owns a **UIView** (root view) and receives lifetime notifications from it
- ❖ Acts as the mediator between the view(s) and the data/logic/model(s)





# The iOS Designer

- ❖ The Xamarin.iOS designer is a visual drag + drop editor for creating and editing screens (View Controllers + Views) in your iOS applications



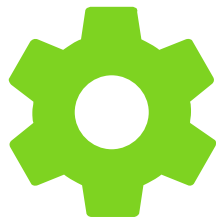


# Parts of the Designer

- ❖ The iOS Designer has several windows which you use to examine, visualize, design the UI of your application



Document  
Outline



Properties  
Explorer



Designer  
Surface



Toolbox



Designer  
Toolbar

# Demonstration

Tour the Xamarin.iOS designer



# Flash Quiz



**Xamarin**  
University

# Flash Quiz

- ① The \_\_\_\_\_ shows a list of views and view controllers that can be dragged onto the storyboard design surface
- a) Toolbox
  - b) Properties Pane
  - c) Designer Toolbar

# Flash Quiz

- ① The \_\_\_\_\_ shows a list of views and view controllers that can be dragged onto the storyboard design surface
- a) Toolbox
  - b) Properties Pane
  - c) Designer Toolbar

# Flash Quiz

- ② A **UIView** is responsible for:
- a) Event publishing
  - b) Visualization
  - c) Managing subviews
  - d) All of the above
  - e) None of the above

# Flash Quiz

- ② A **UIView** is responsible for:
- a) Event publishing
  - b) Visualization
  - c) Managing subviews
  - d) All of the above
  - e) None of the above



# Storyboards vs. XIBs

❖ iOS supports two designer file formats

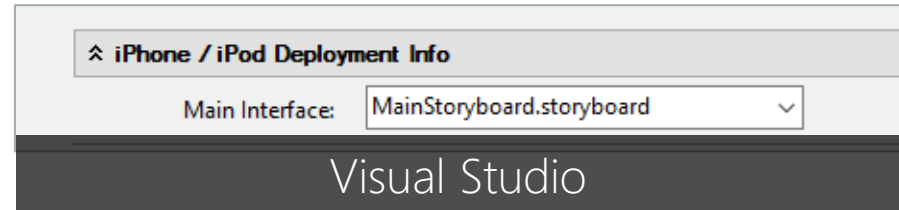
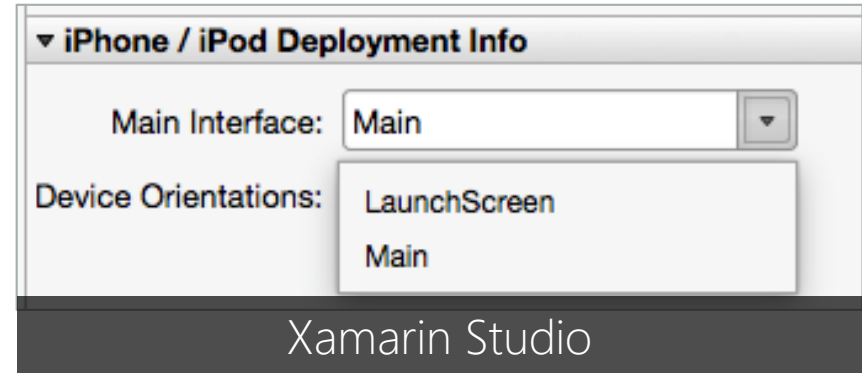
**Storyboards** let you design multiple screens together with the relationships between them; this is the default file created for your app



**XIB** is the original format which defines a single screen or part of a screen; this is used today for the Launch Screen

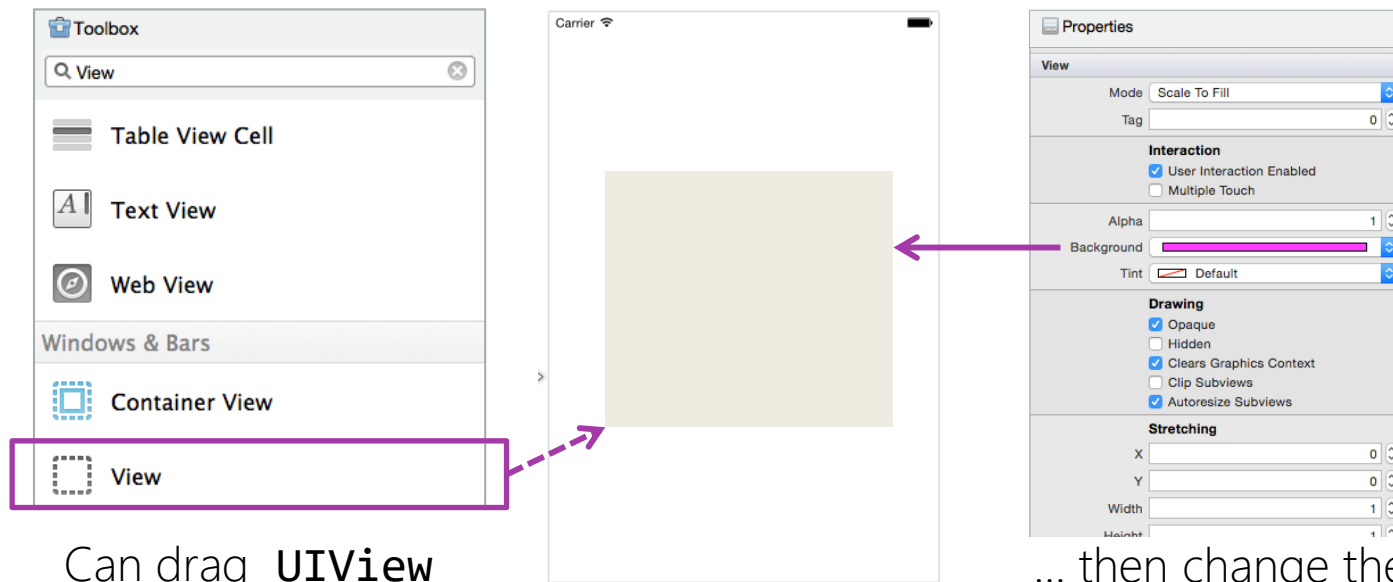
# Using Storyboards

- ❖ Most of the time you will only have one Storyboard in your app, but you can add as many as you need to segregate or share your UI definitions
- ❖ **Info.plist** identifies the one to start the app with and is editable under **iOS Application** settings



# Workflow [Xamarin Studio]

- ❖ Drag widgets onto the designer surface and then set properties

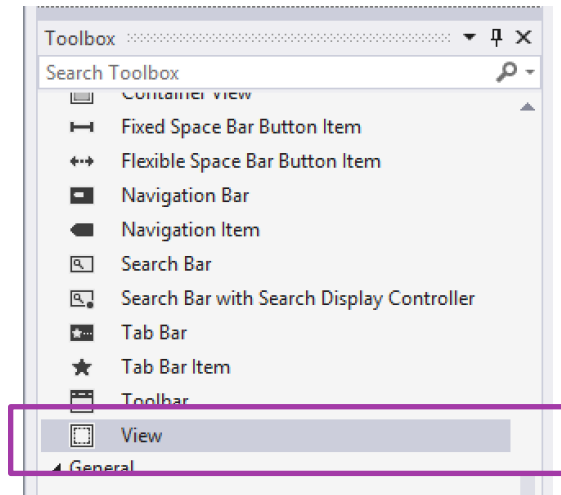


Can drag **UIView**  
from the **toolbox**

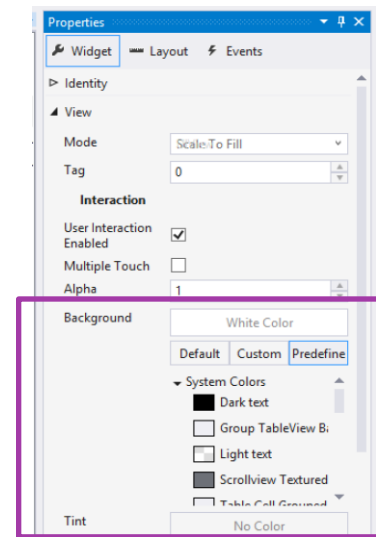
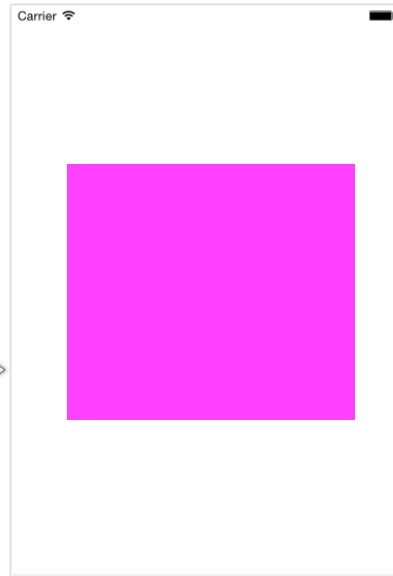
... then change the  
background color

# Workflow [Visual Studio]

- ❖ Drag widgets onto the designer surface and then set properties



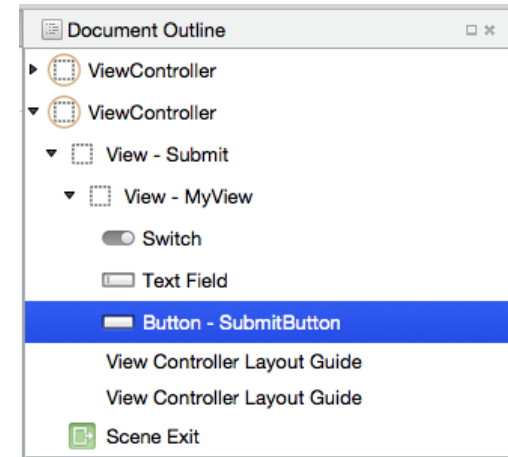
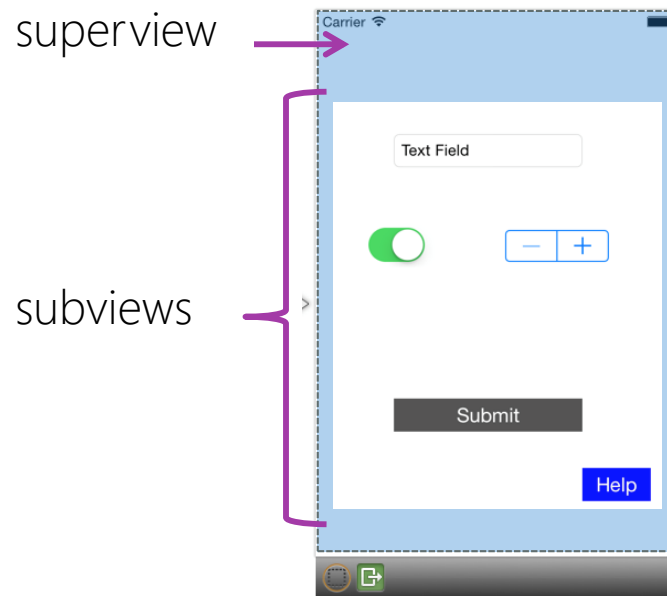
Can drag **UIView**  
from the **toolbox**



... then change the  
background color

# Layout and subviews

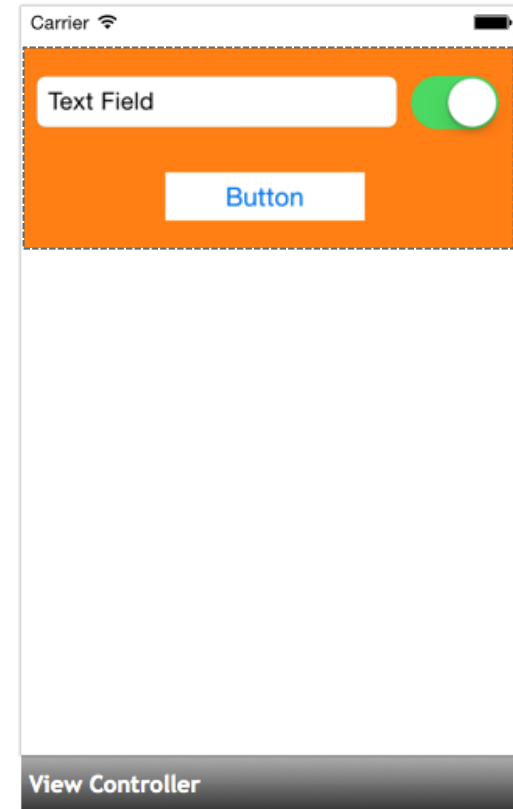
- ❖ Controls (subviews) can be positioned onto the root **UIView** (superview)



Use the document outline view to see relationships

# Composite controls

- ❖ Can take advantage of the view architecture to create *composite controls* by nesting controls within a **UIView**
- ❖ Composite controls can be made reusable and are easily moved or animated as a group by adjusting the parent view



# Individual Exercise

Create the UI for a single view application



# Summary

1. Describe the iOS Designer
2. Identify controls and properties
3. Demonstrate the designer workflow
4. Work with subviews



# Describe and use Auto Layout

# Tasks

1. Describe the Auto Layout system
2. Identify constraints
3. Add constraints using the Designer



# Responsive interface design

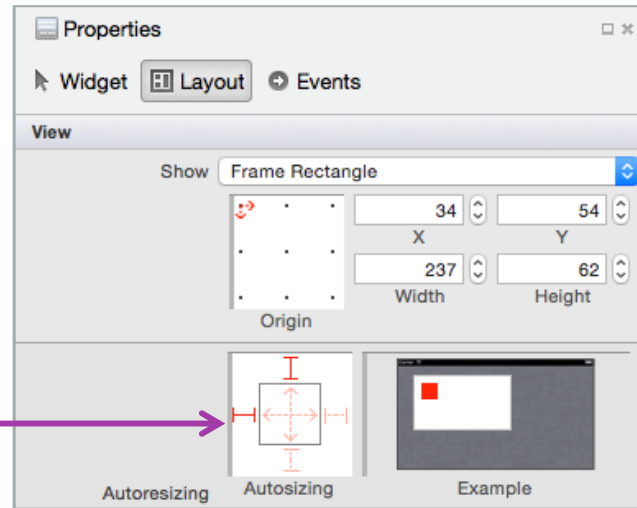
- ❖ There are several things which can affect the layout of your UI at runtime
  - Orientation changes
  - Running on a device with a different resolution or form factor
  - Positioning dynamic content
  - Working with user-selectable fonts
  - Localization



# Layout solutions

- ❖ Apple has two APIs to manage layout rules in the UI design

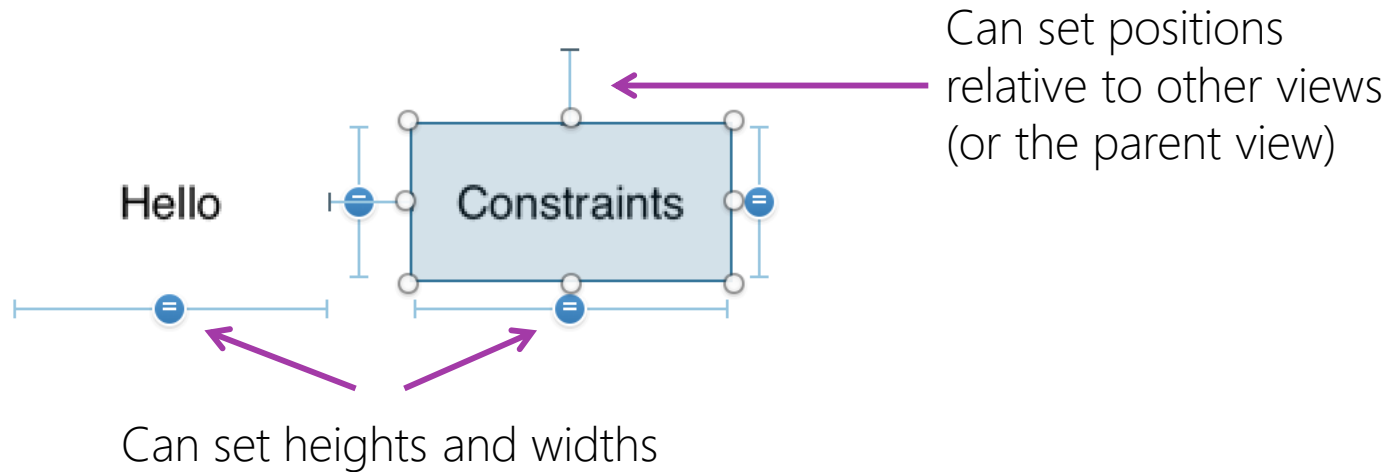
You define each side of the frame with either a "flexible" or "fixed" margin to decide if it stretches with, or is pinned to the parent



Autosizing Masks

# What is Auto Layout?

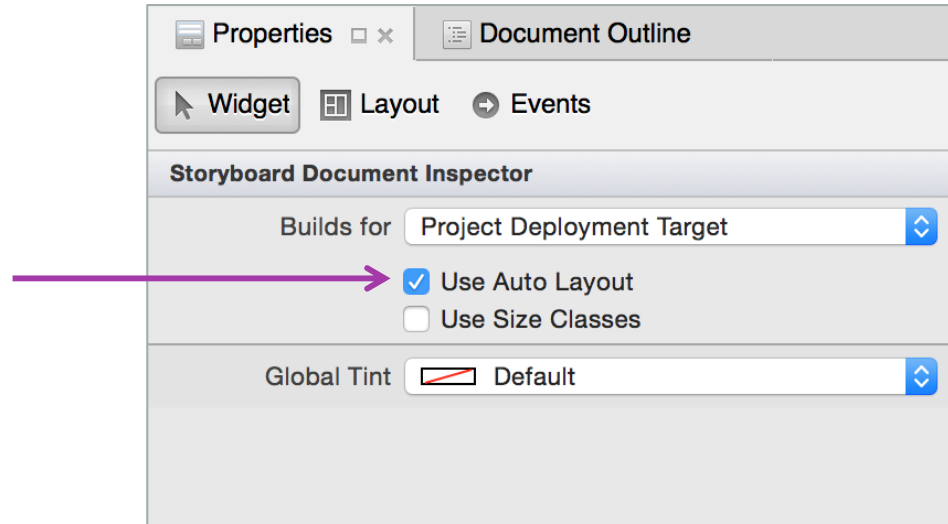
- ❖ **Auto Layout** is a system that helps organize the application UI by describing relationships between visual elements



# Auto Layout in the Designer

- ❖ Storyboard designer allows us to visually manage Auto Layout constraints without writing any code

Enabled by default but can be turned on and off in the Storyboard properties in Xamarin Studio or Visual Studio – this will cause the Storyboard to revert back to the "Springs and Struts" approach





# What are Constraints?

- ❖ Constraints determine *one aspect* of a **UIView** position or size and essentially **form the rules** that describe the layout

```
view.left = superview.left
```

```
view.top = superview.top + 20
```

```
view.top = otherView.bottom + 8
```

```
view.height = 0.5 * superview.height
```

```
view.height = 0.5 * view.width
```

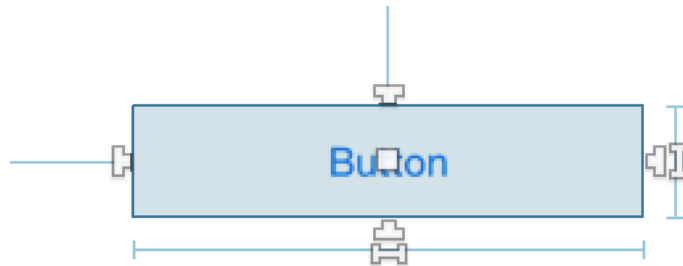
# Constraint behavior

- ❖ Constraints are:
  - Applied to views
  - Cumulative
  - Prioritized
  - Able to cross views
- ❖ They decide the position and size of the **UIView** they are applied to



# Xamarin.iOS Designer

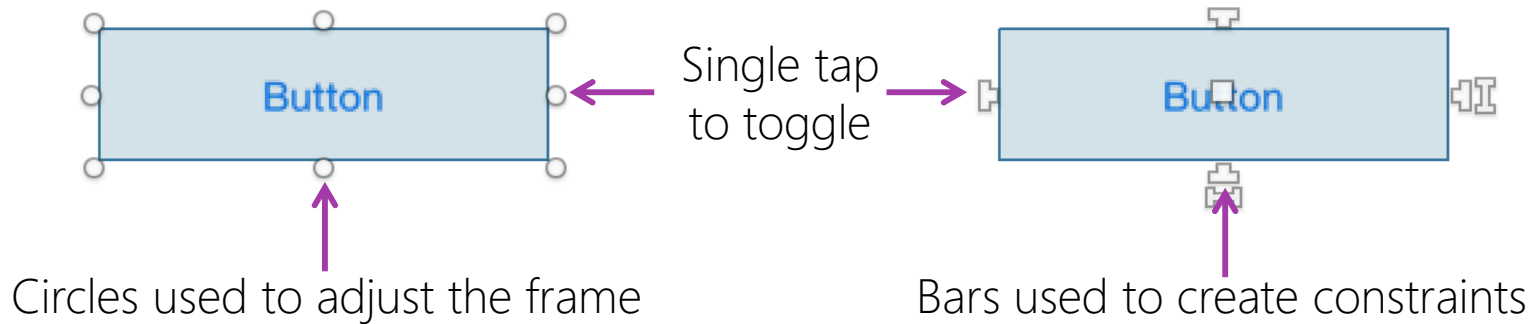
- ❖ The designer adds the constraints directly into the Storyboard and iOS will then apply them when the UI is inflated at runtime



constraints are shown graphically and can be changed either on the designer surface or in the property pad

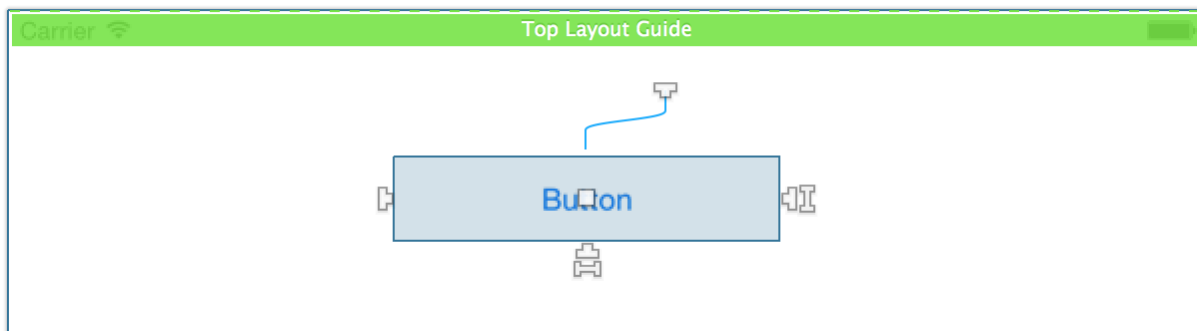
# Constraint "Mode"

- ❖ In the iOS Designer, **single-tap** views to toggle between editing the frame and editing constraints



# Adding Constraints

- ❖ Use the *dragging control decorators* on a view to create a constraint with itself, the parent, or a sibling view




Can select and drag the handles and drop onto the target view

# Types of Constraints

- ❖ Designer supports manipulation of three types of constraints

A green parallelogram shape, tilted to the right, containing the text 'Spacing Constraints' in white.

Spacing  
Constraints

A blue parallelogram shape, tilted to the right, containing the text 'Sizing Constraints' in white.

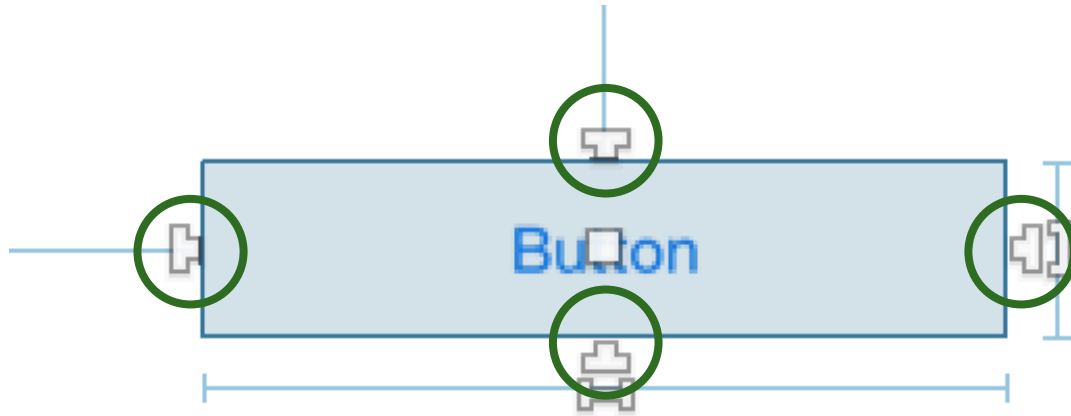
Sizing  
Constraints

A purple parallelogram shape, tilted to the right, containing the text 'Alignment Constraints' in white.

Alignment  
Constraints

# Spacing constraints

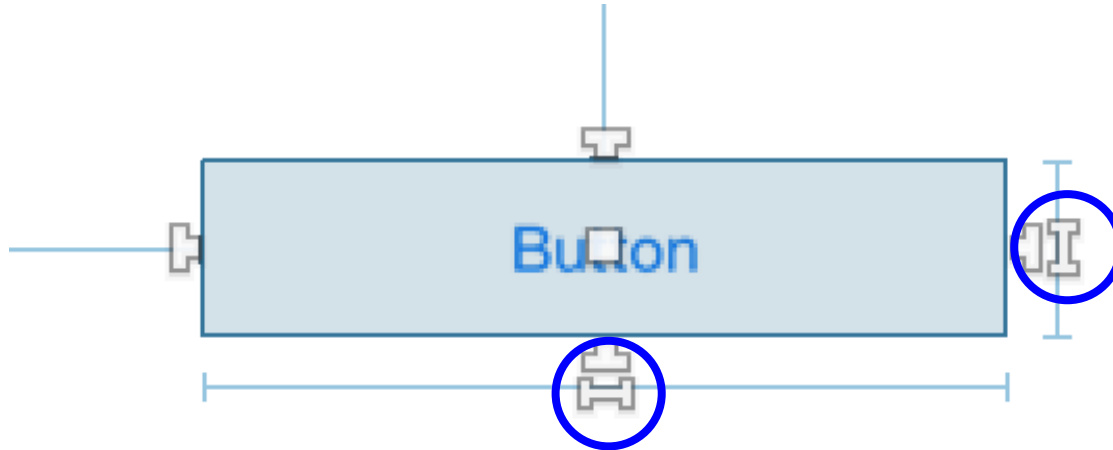
- ❖ **Spacing constraints** allow you to position a view relative to another view (or parent) by dragging the T-handle shapes on each edge





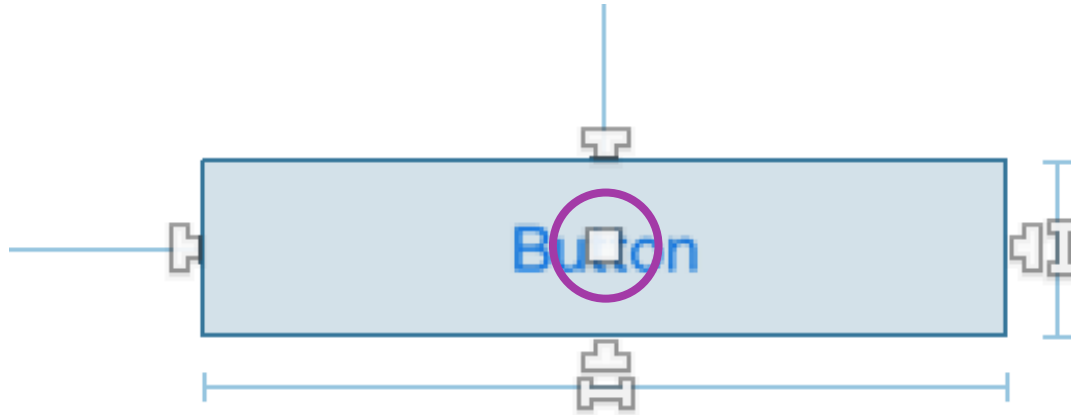
# Sizing constraints

- ❖ **Sizing constraints** allow you to control a views width and height (can be a constant, fixed to another constraint, or an inequality) by dragging the center "I" bar shape on the right and bottom edge of the view



# Alignment constraints

- ❖ **Alignment constraints** allow you to align a view to the X or Y axis of it's superview or a sibling



# Flash Quiz



**Xamarin**  
University

# Flash Quiz

- ① Auto Layout \_\_\_\_\_
- a) Is only available in the Designer
  - b) Describes relationships between visual elements
  - c) Must be used and cannot be turned off

# Flash Quiz

- ① Auto Layout \_\_\_\_\_
- a) Is only available in the Designer
  - b) Describes relationships between visual elements
  - c) Must be used and cannot be turned off

# Flash Quiz

- ② Spacing constraints are used to position a view
- a) True
  - b) False

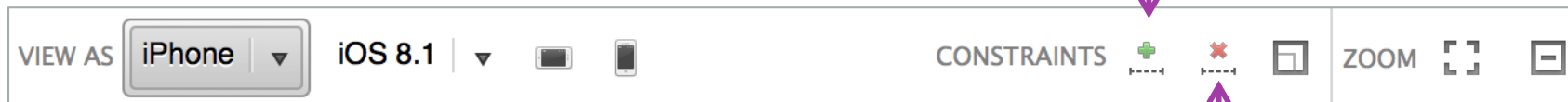
# Flash Quiz

- ② Spacing constraints are used to position a view
- a) True
  - b) False

# Add Recommended Constraints

- ❖ Xamarin Designer can **add recommended constraints** to a View

Adds 4 constraints to set position and size

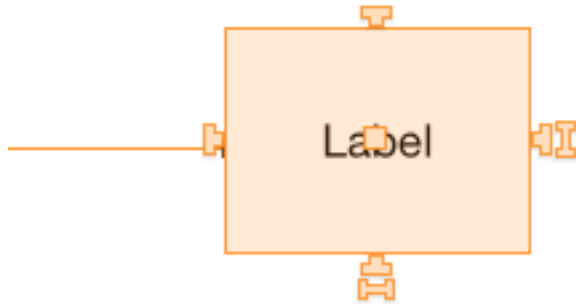


Removes all constraints  
for a selected view

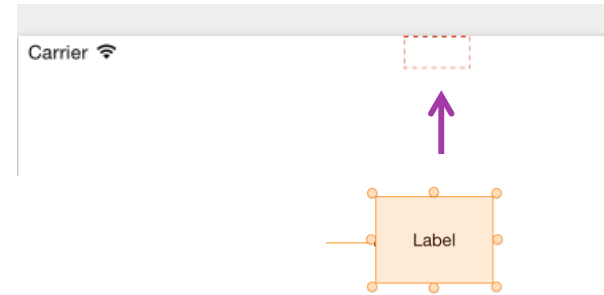


# What is a fully-constrained view?

- ❖ A *fully-constrained view* has enough constraints to uniquely describe the view's position and size, typically this requires **4 constraints**



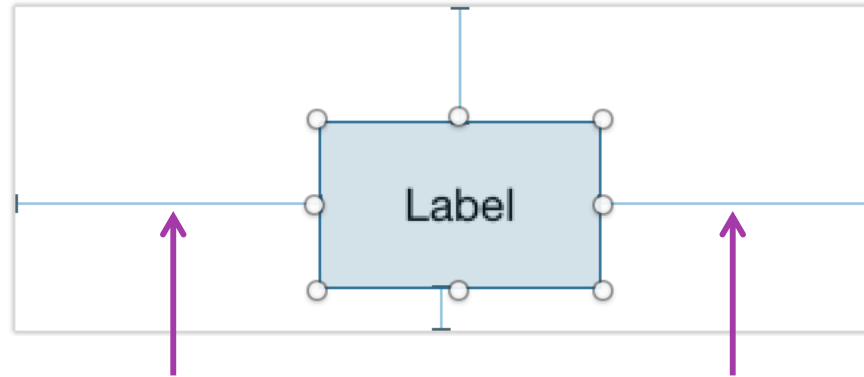
Designer shows view in orange when it does not have enough constraints to determine size and/or position



... will often show a dotted rectangle for where the view will be positioned/sized at runtime

# What is a fully-constrained view?

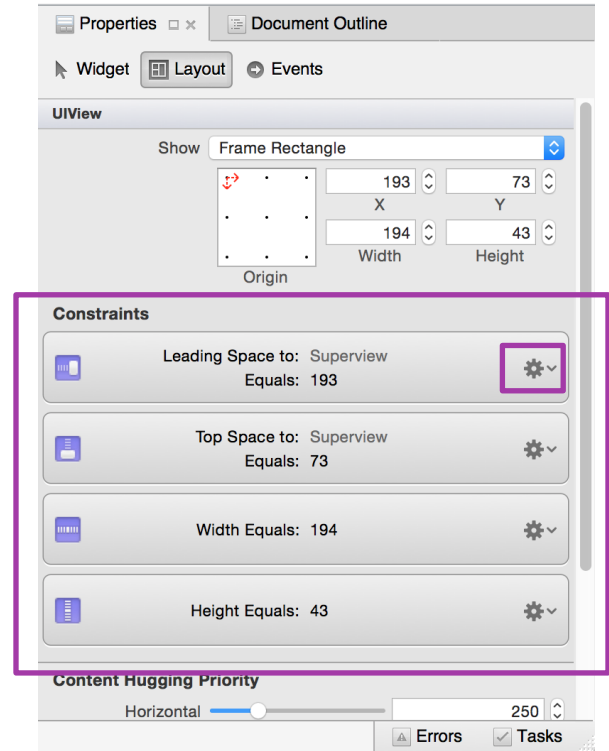
- ❖ A *fully-constrained view* has enough constraints to uniquely describe the view's position and size, typically this requires **4 constraints**



Properties can be constrained indirectly. We can constrain the left edge and right edge which will effectively constrain the width.

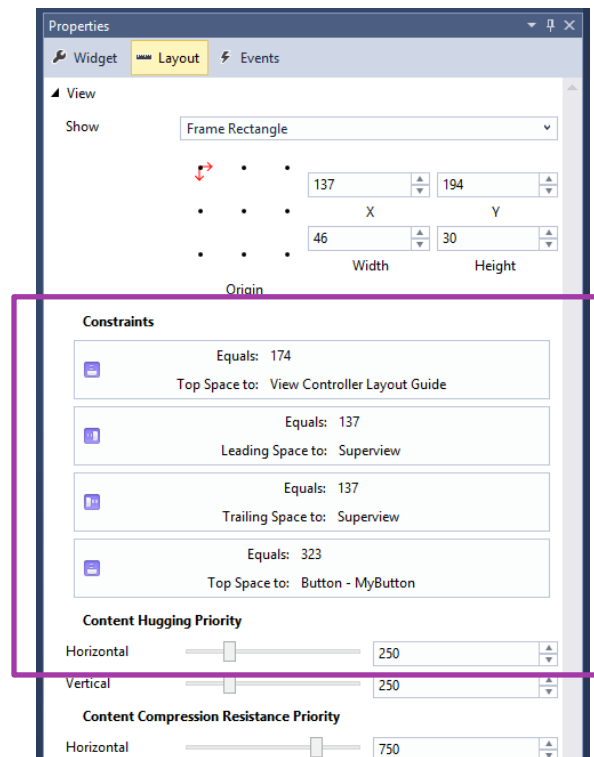
# Editing Constraints

- ❖ **Layout Area in the Properties Pane**  
provides a more powerful way to edit and manage constraints
  - Provides an overview of all constraints
  - Can “fine-tune” constraints through an inline editor



# Editing Constraints [Visual Studio]

- ❖ **Layout Area in the Properties Pane** provides a more powerful way to edit and manage constraints
  - Provides an overview of all constraints
  - Can “fine-tune” constraints through an inline editor



# Group Exercise

Add constraints to the fireworks app



**Xamarin**  
University

# Summary

1. Describe the Auto Layout system
2. Identify constraints
3. Add constraints using the Designer



# Interact with controls and views programmatically

# Tasks

1. Associate a class for the **UIViewController**
2. Adding actions to a control
3. Naming views
4. Inspect outlets and actions





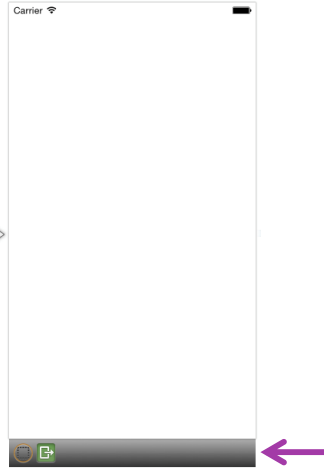
# User interaction

- ❖ Applying behavior to the views of an app is essential if you want your code to respond to user interactions
- ❖ Keep in mind that not every view needs to be accessible in code – only things that drive the business logic or display results

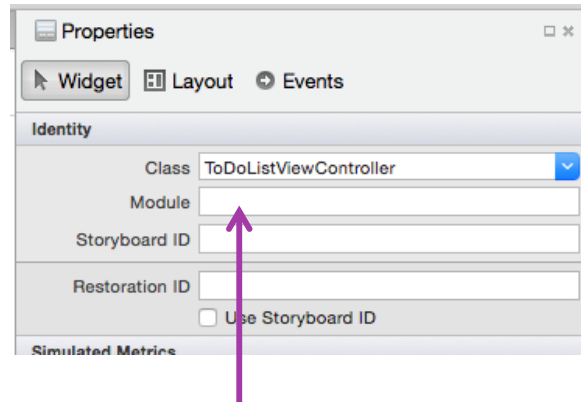


# Assign a class

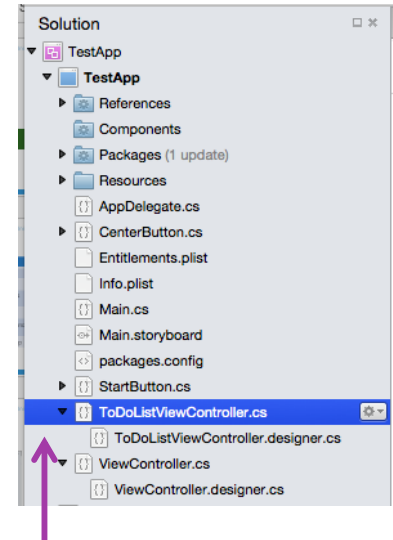
- ❖ In order to apply behaviors to your controls the **UIViewController** must have a class associated to it



Select the grey bar  
on the view controller



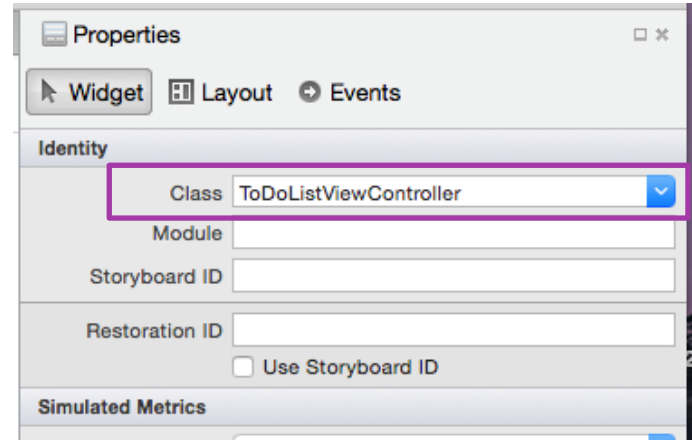
Assign it a class name



Xamarin will populate a  
C# file in the solution

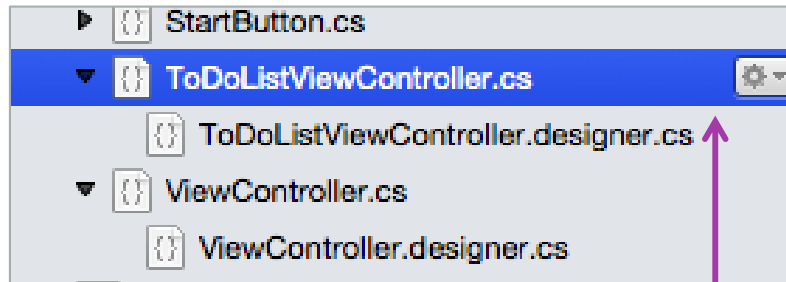
# Naming guidelines

- ❖ There are two recommended practices to follow when naming your backing class:
  - 1) Name should reflect what the screen *does* or *manages* this makes it easier to identify when you have multiple screens
  - 2) Name should end with the "**ViewController**" suffix to make it obvious what it is



# Partial classes [main file]

- ❖ When you assign a class in the designer, the class will be marked as a partial class and split into two files



```
5
6 namespace TestApp
7 {
8     partial class ToDoListViewController : UIViewController
9     {
10         public ToDoListViewController (IntPtr handle) : base
11         {
12         }
13     }
14 }
15
```

The .cs file is where you will code the behaviors for your view controller

# Partial classes [designer file]

- ❖ When you assign a class in the designer, the class will be marked as a partial class and split into two files



```
12 namespace TestApp
13 {
14     [Register ("ToDoListViewController")]
15     partial class ToDoListViewController
16     {
17         [Outlet]
18         [GeneratedCode ("iOS Designer", "1.0")]
19         UIButton MyButton { get; set; }
20
21         [Action ("MyButton_TouchUpInside:")]
22         [GeneratedCode ("iOS Designer", "1.0")]
```

The **designer.cs** file is a representation of the storyboard for the compiler, you should never change this file directly since it is auto-generated

# Registering a class with iOS

- ❖ Classes that will be instantiated by iOS need to be *registered* with the Objective-C runtime – this is done through a **[Register]** attribute

```
12 namespace TestApp
13 {
14     [Register ("ToDoListViewController")]
15     partial class ToDoListViewController
16     {
17         [Outlet]
18         [GeneratedCode ("iOS Designer", "1.0")]
19         UIButton MyButton { get; set; }
20
21         [Action ("MyButton_TouchUpInside:")]
22         [GeneratedCode ("iOS Designer", "1.0")]
```

# View Controller constructor

- ❖ iOS uses a custom constructor to create the View Controller

```
public partial class ViewController : UIViewController
{
    public ViewController(IntPtr handle) : base(handle)
    {
    }

    ...
}
```

Must have this constructor and chain to the base if iOS is going to instantiate this View Controller (e.g. load from a Storyboard)

# Flash Quiz



**Xamarin**  
University



# Flash Quiz

- ① The **.designer.cs** file
- a) Is for coding a view's behavior
  - b) Is a representation of the storyboard in code
  - c) All of the above
  - d) None of the above

# Flash Quiz

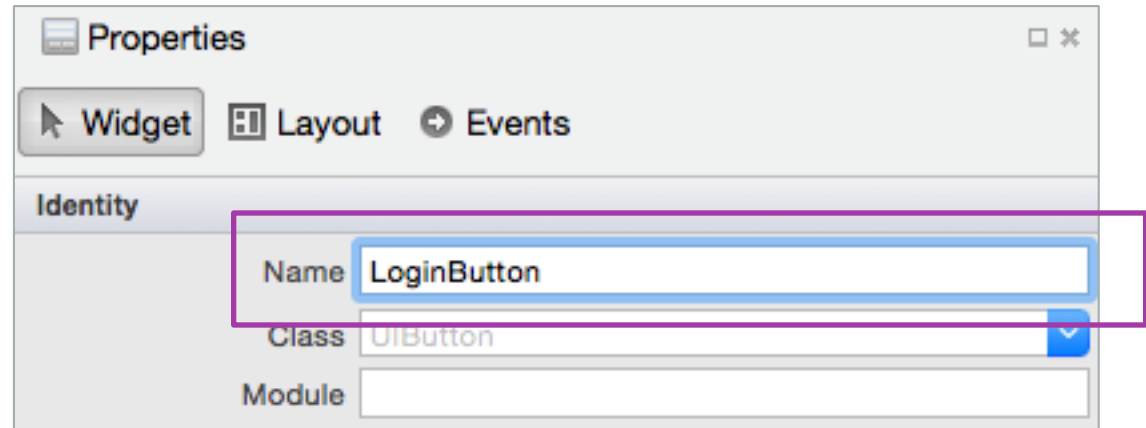
- ① The **.designer.cs** file
- a) Is for coding a view's behavior
  - b) Is a representation of the storyboard in code
  - c) All of the above
  - d) None of the above

# Name your view

- ❖ Can name your views to make them accessible to View Controller code



Select the control in the design surface and then set the Name



**Hint:** as with naming View Controllers, it is advisable to use a name which shows the purpose and then the type – to make it easier to identify in code

# What is an Outlet?

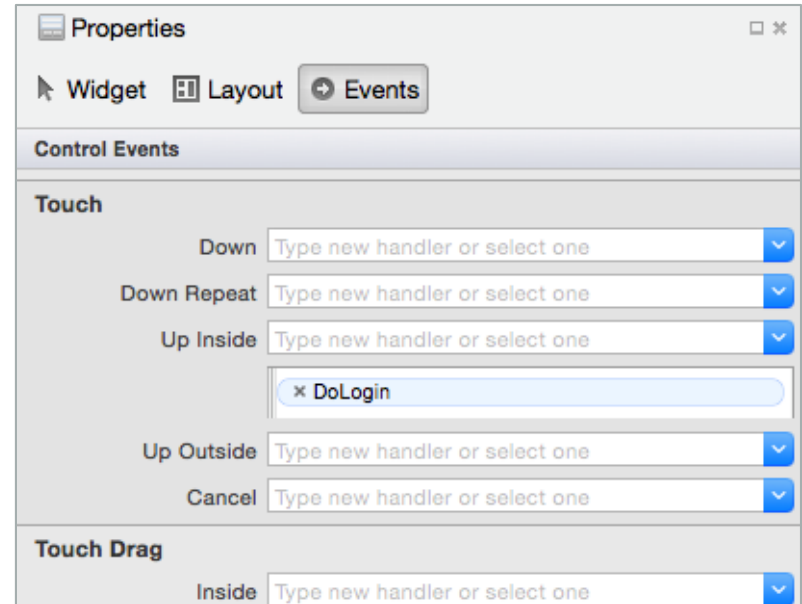
- ❖ An **outlet** is a property that is tied to a control in the UI design
  - Control must be defined in the screen owned by the View Controller
  - Property is **private** to the View Controller class
  - Decorated with an **[Outlet]** attribute to register it with iOS

```
[Register ("TodoListViewController")]
partial class TodoListViewController
{
    [Outlet]
    [GeneratedCode ("iOS Designer", "1.0")]
    UIButton LoginButton { get; set; }
    ...
}
```

Designer adds this code to your **designer.cs** file when you name a control in the storyboard

# What is an Action?

- ❖ Actions are methods that are called by a view in response to a runtime interaction or event
- ❖ In the Designer you can choose **Events** on the properties pane and associate methods to the actions the selected view raises at runtime
- ❖ Can double-click on most controls to add a handler for the "default" action



# Implementing Actions

- ❖ Actions wired up in the designer are mapped to partial methods defined in the designer portion of your View Controller class and implemented in your main source file

ViewController.designer.cs

```
[Action ("DoLogin:")]  
[GeneratedCode ("iOS Designer", "1.0")]  
partial void DoLogin (UIButton sender);
```

```
partial void DoLogin(UIButton sender) {  
    // TODO: add logic here  
}
```

ViewController.cs

# Individual Exercise

Code behaviors for your app



**Xamarin**  
University

# Summary

1. Associate a class for the **UIViewController**
2. Identify partial methods
3. Name views
4. Inspect outlets and actions

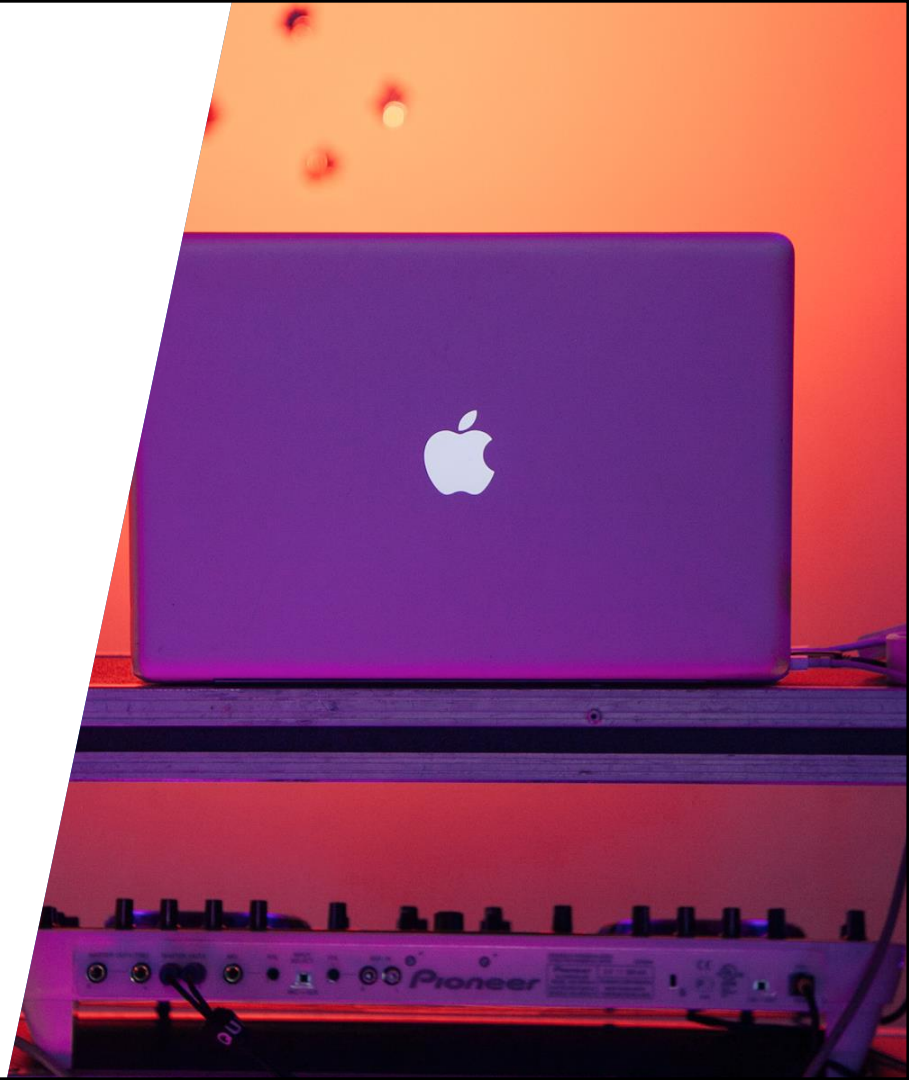




# Apply navigation using segues

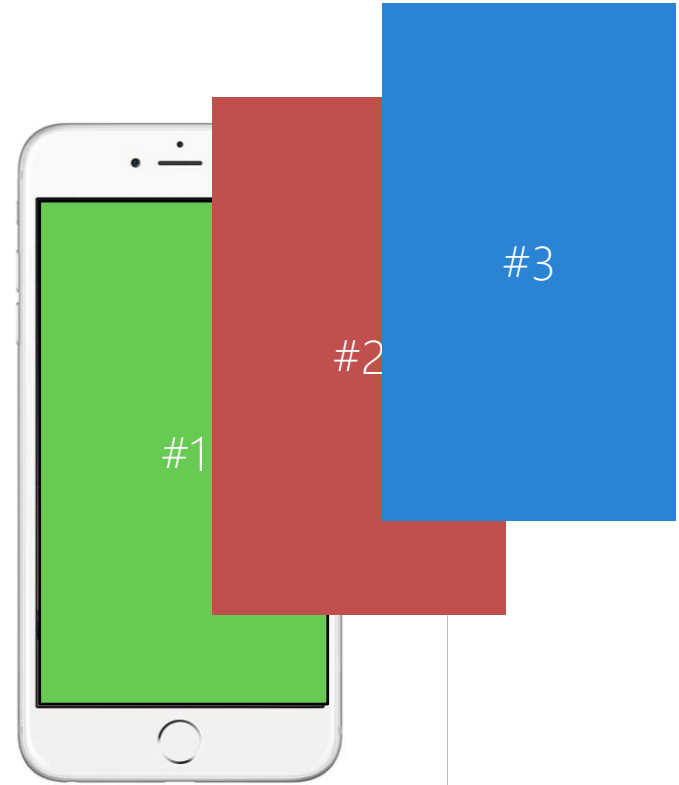
# Tasks

1. Add a second **UIViewController**
2. Code a button to present a view controller
3. Code a button to dismiss a view controller
4. Utilize segues to create navigation



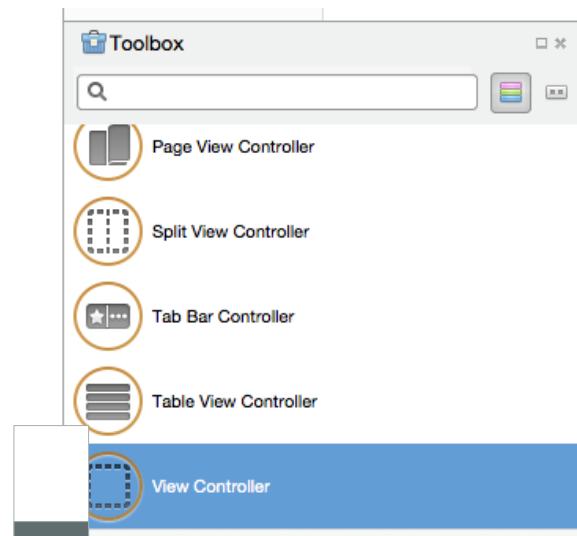
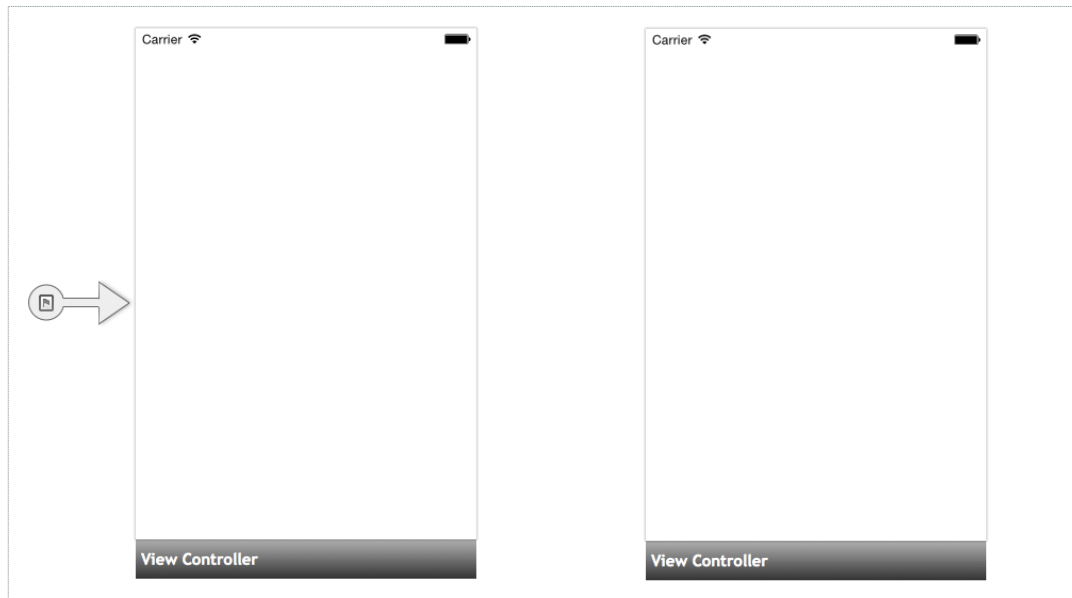
# Multi-screen apps

- ❖ Most applications consist of more than one screen
- ❖ Can define multiple screens in the Storyboard
- ❖ Can then display secondary screens through code, or by defining the relationships in the designer



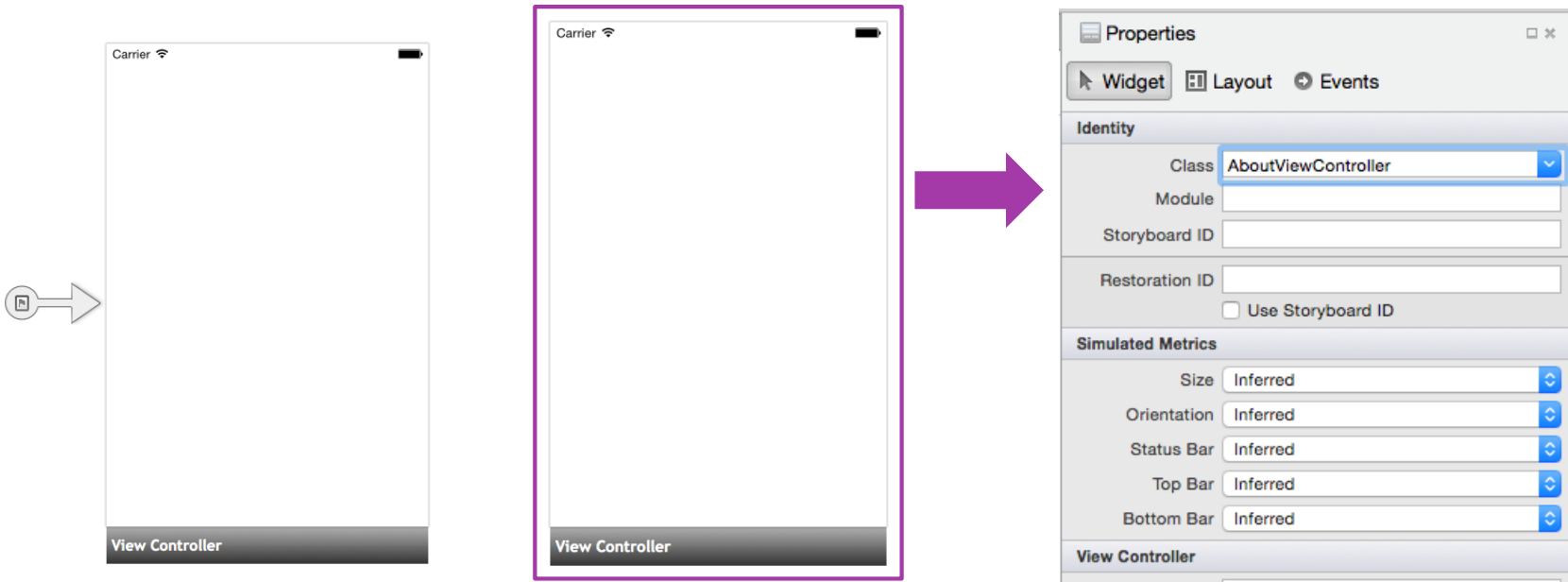
# Adding screens to the storyboard

- ❖ You can add screens to your app by dragging a view controller onto the storyboard and then add your **UIViews** onto it



# Create a class

- ❖ All new view controllers added to the storyboard should be assigned a backing class



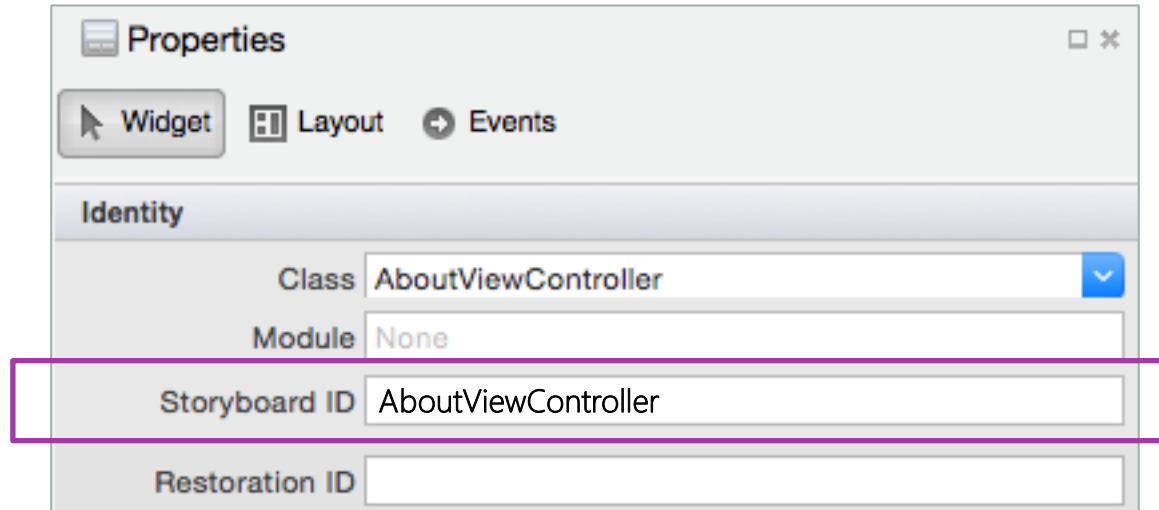
# Instantiating a View Controller

- ❖ View Controllers defined in Storyboards must be created through the Storyboard APIs to get the proper views created

```
partial void ShowAboutPage(UIButton sender) {  
  
    UIStoryboard storyboard = this.Storyboard;  
    AboutViewController viewController = (AboutViewController)  
        storyboard.InstantiateViewController("AboutViewController");  
    ...  
}
```

# Naming a View Controller

- ❖ Must set the Storyboard Id on the View Controller to identify it to the Storyboard from code – a good practice is to give it the same name as the class that defines it in code



# Present the view controller

- ❖ Can use the **PresentViewController** method to display a new View Controller in a modal fashion on top of your existing screen

```
partial void ShowAboutPage(UITableView sender)
{
    UIStoryboard storyboard = this.Storyboard;
    AboutViewController viewController = (AboutViewController)
        storyboard.InstantiateViewController("AboutViewController");

    this.PresentViewController(viewController, true, null);
}
```



# Dismiss a modal view controller

- ❖ To return to the previous View Controller, use the **DismissViewController** method in your active view controller

```
partial class AboutViewController : UIViewController
{
    ...
    partial void OnGoBack(UITableView sender)
    {
        this.DismissViewController(true, null);
    }
}
```

# Changing the transition style

- ❖ Can customize the animation used to transition to the new controller through the **ModalTransitionStyle** property

```
partial void ShowAboutPage(UIButton sender)
{
    AboutViewController viewController = ...;
    viewController.ModalTransitionStyle =
        UIModalTransitionStyle.PartialCurl;

    this.PresentViewController(viewController, true, null);
}
```

- F CoverVertical
- F CrossDissolve
- F FlipHorizontal
- F PartialCurl



# Individual Exercise

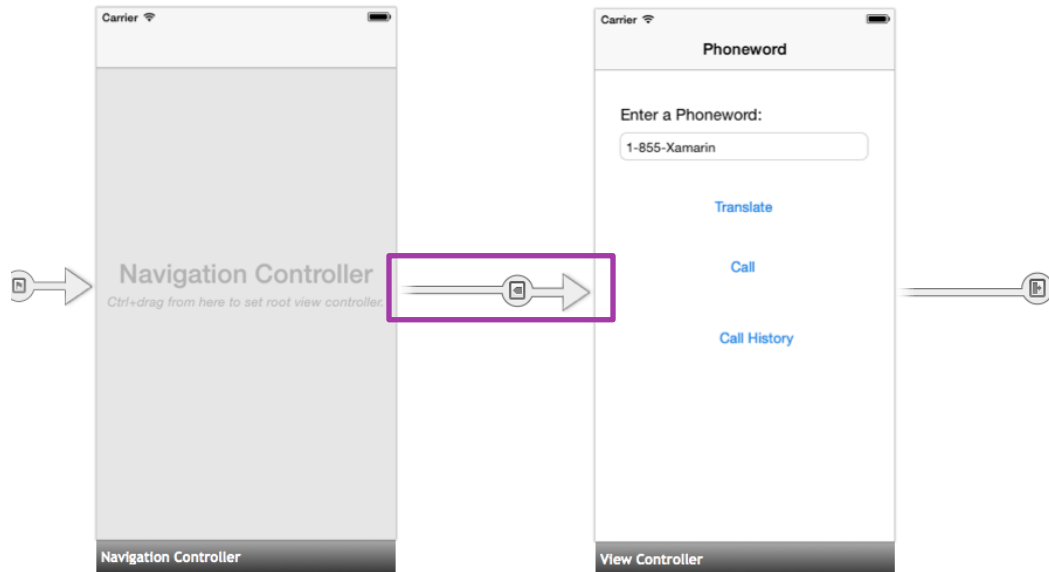
Add a second screen to your app  
and code a button to navigate to it



**Xamarin**  
University

# What is a Segue?

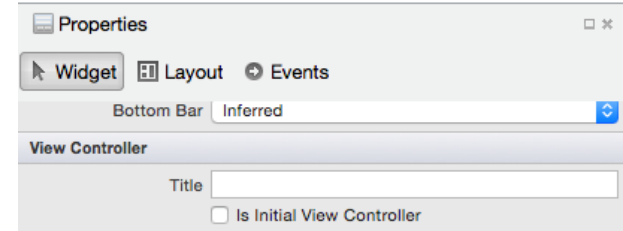
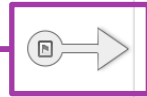
- ❖ *Segues* ("segways") define the transitions between the screens of our app in the designer



# Sourceless segue

- ❖ The **sourceless segue** indicates the root (initial) view controller

Click+Drag to move the **sourceless segue** to a different screen

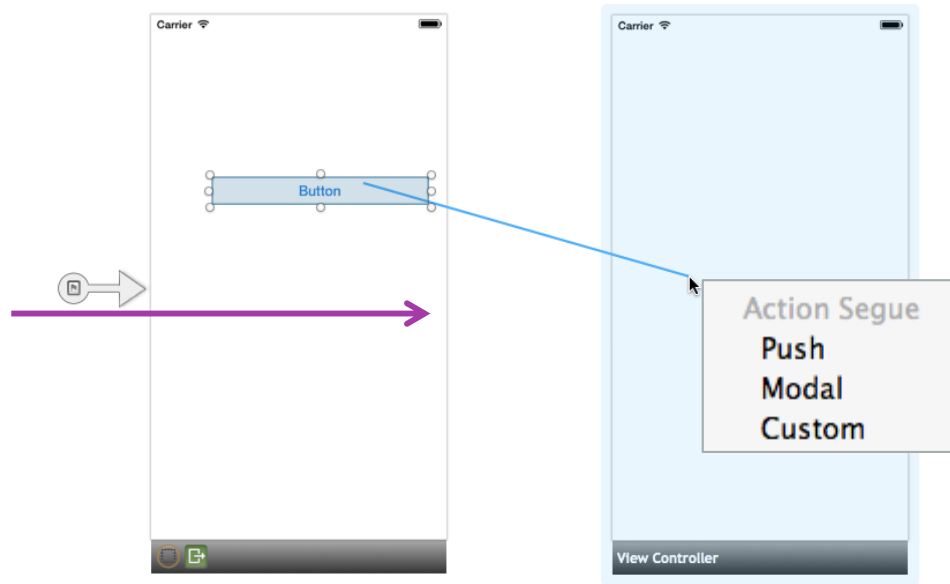


... or select the view controller and check the initial view controller checkbox

# Create a segue relationship

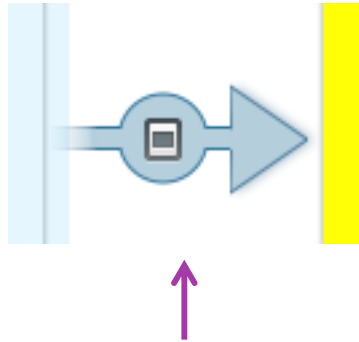
- ❖ Use **Ctrl+Drag** to create segues between two screens

The blue connector appears as you drag your mouse from a control to the target screen

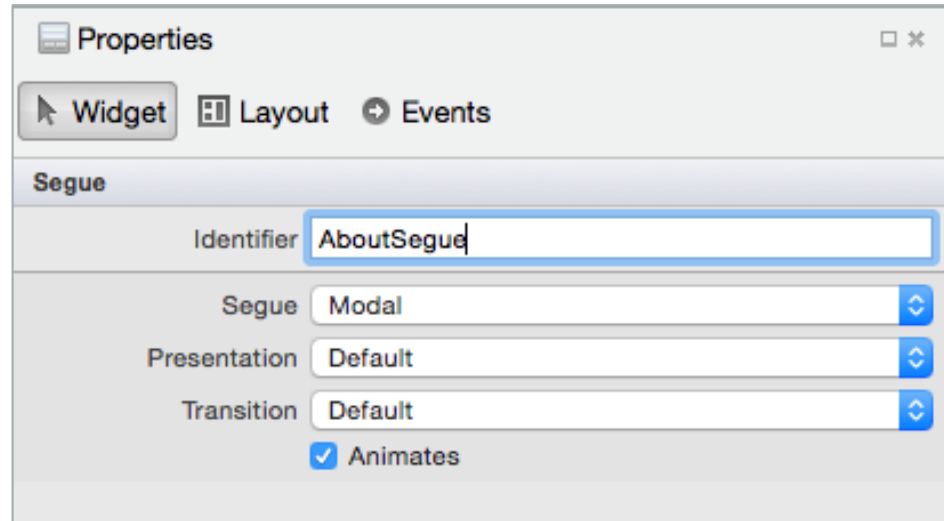


# Segue properties

- ❖ Segues have properties such as an ID and transition type



Select the segue on the storyboard to view segue options




# Relationship types

- ❖ There are two types of relationships that can be created:



Action



Manual

Action segues are defined between an active view such as a button and a screen – these can be trigger the segue directly

Manual segues are defined between a non-active view or view controller and a screen – these must be activated in code



# Run a segue from code

- ❖ Can use **PerformSegue** in a View Controller to initiate a segue from code – this allows you to define the transition in the Storyboard, but decide when to run it based on your application logic

```
partial void ShowAboutPage(UITableView sender)
{
    this.PerformSegue("AboutSegue", this);
}
```

Takes the identifier of the segue

.. And the sender

# Stopping a segue

- ❖ Sometimes you need to stop a segue from occurring due to some application state

```
public override bool ShouldPerformSegue(  
    string segueIdentifier, NSObject sender)  
{  
    if (IsScreenDataValid()  
        && segueIdentifier != "AboutSegue")  
        return false; // do not run any segue except About  
  
    return true; // allow segue  
}
```

# Influence a segue

- ❖ Sometimes you need to just setup the target screen with some data from the source – can use **PrepareForSegue** override

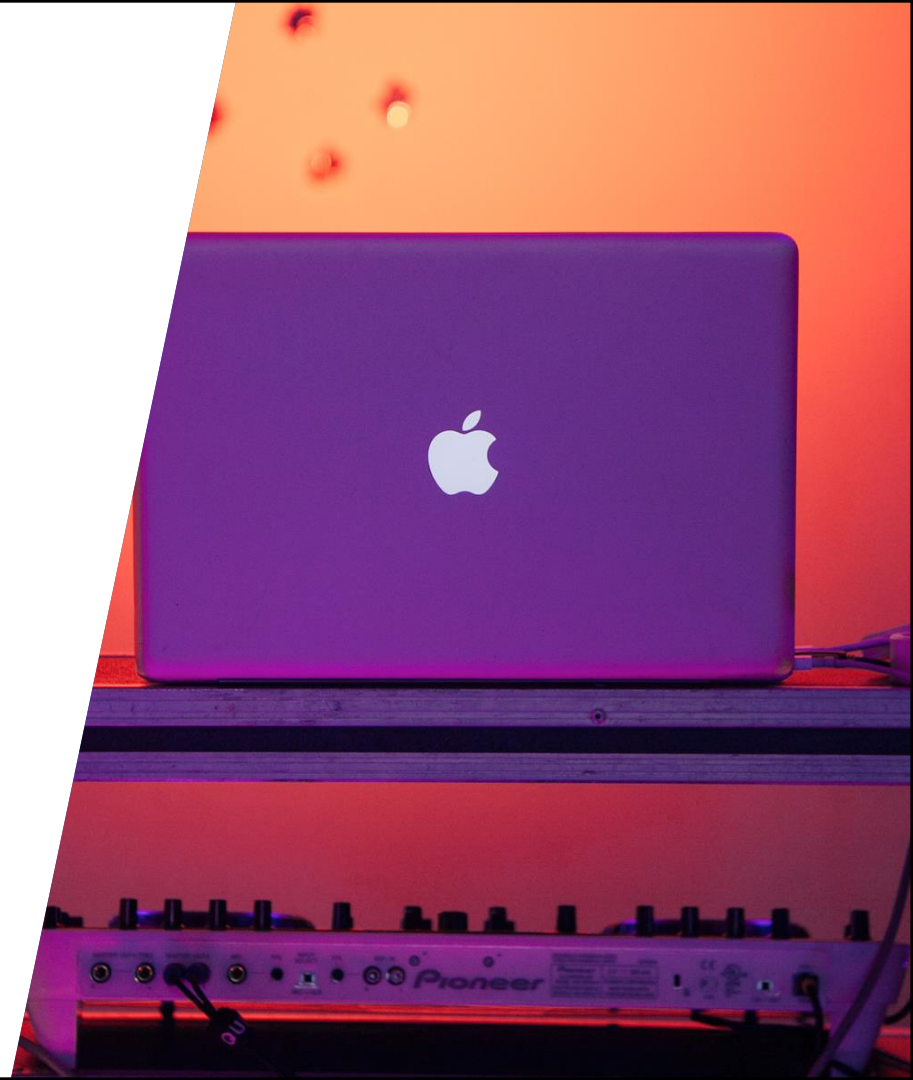
```
public override void PrepareForSegue(UIStoryboardSegue segue,
                                     NSObject sender)
{
    if (segue.Identifier == "AboutSegue") {
        var vc = segue.DestinationViewController as
            AboutViewController;
        vc.RegisteredUserName = ...; // Some custom property
    }
}
```

# Individual Exercise

Add segues to define the navigation

# Summary

1. Add a second **UIViewController**
2. Code a button to present a view controller
3. Code a button to dismiss a view controller
4. Utilize segues to create navigation



# Thank You!

Please complete the class survey in your profile:  
[university.xamarin.com/profile](https://university.xamarin.com/profile)

