HACKING WITH SWIFT

HACKING WITH tvOS

COMPLETE TUTORIAL COURSE

Learn to make tvOS apps with real-wo
Swift projects

FREE SAMPLE

Paul Hudson

Project 1
Randomly Beautiful

Setting up

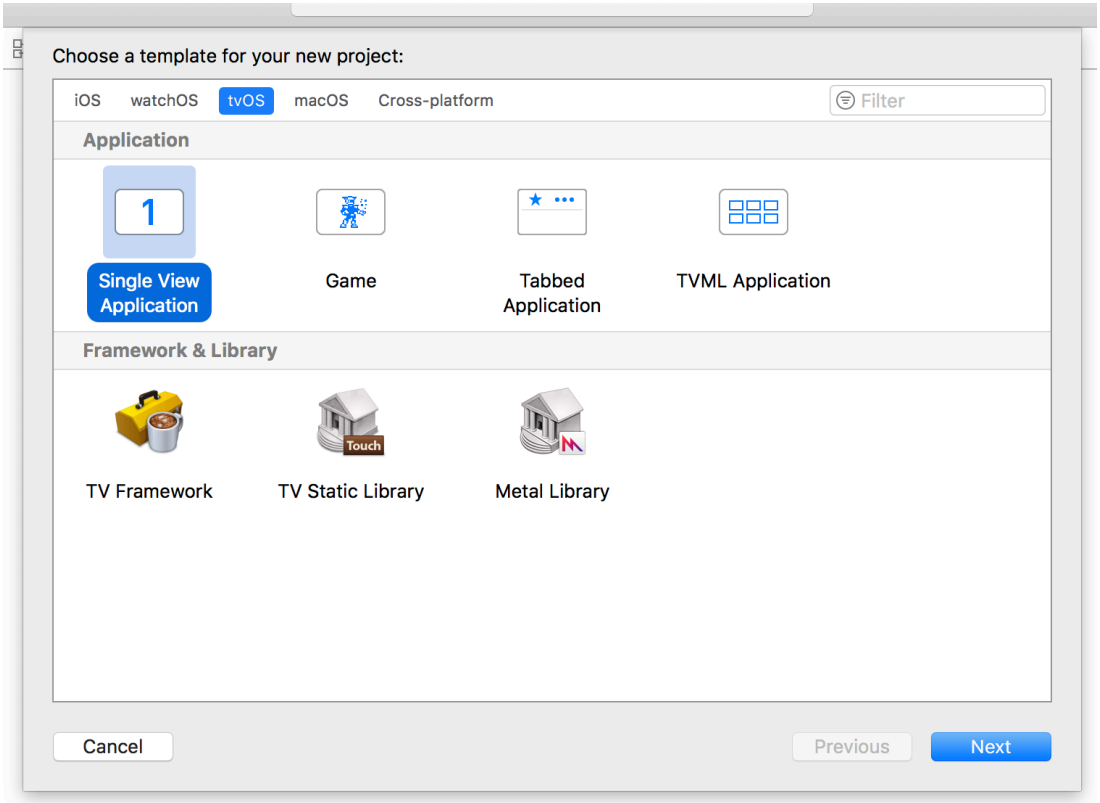
In this first project we're going to build a replacement for the built-in Apple TV screensaver. By default, you get slow-motion flyovers of several locations around the world, and I have to admit it's quite beautiful. But as with most things, the novelty wears off sooner or later, and that's where this app comes in.

To make this work you're going to meet some of the fundamental tvOS components: creating screens of data using **UIViewController**, displaying images using **UIImageView**, showing scrolling lists of data using **UITableView**, and even a little animation to make our images crossfade.

But there's more. As I've said previously, tvOS is a platform of *consumption*, which means that learning to fetch and parse remote data is important. So, we're going to start with that now, and use it several times in future projects so you have lots of time to get the hang of it.

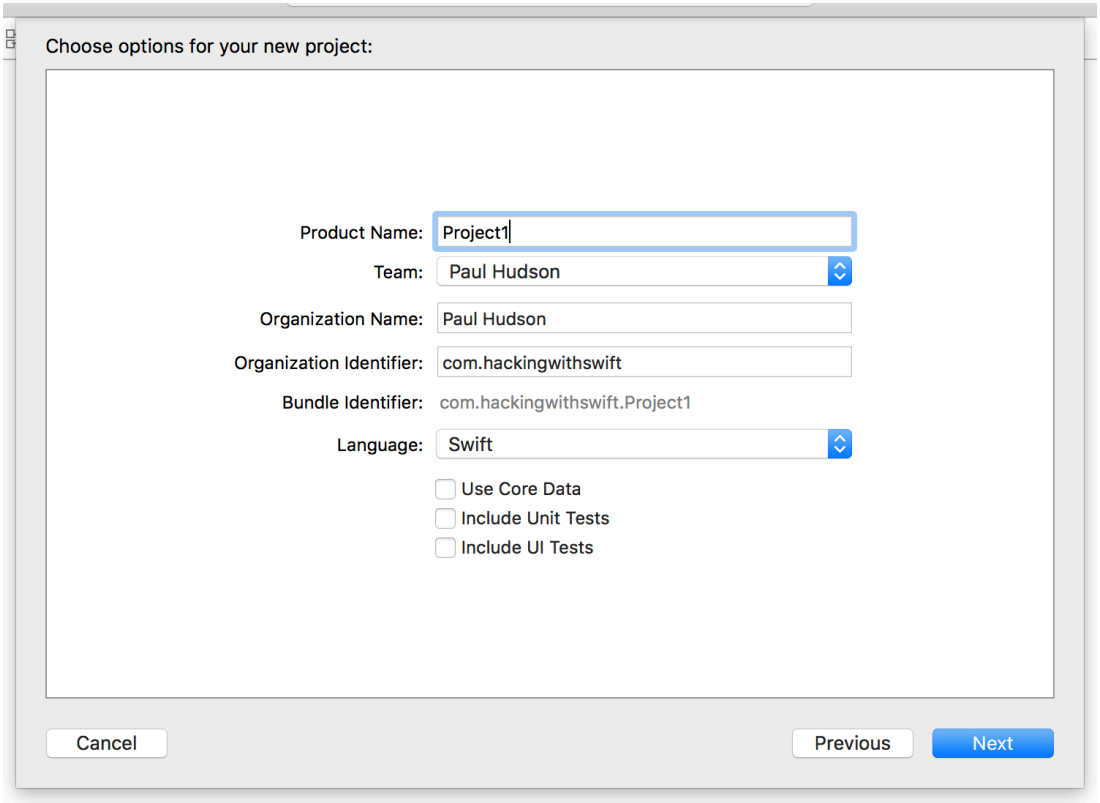
This means I need to teach you two techniques that normally I would leave until later on: how to parse JSON data, and how to run code in the background. First projects are always long and slow because you have so many fundamental techniques to learn just to get moving, but with those extra topics added on top I've had to work extra hard to structure this project so you learn new things in a slow, structured way.

Let's get started now: launch Xcode and create a new tvOS project using the Single View App template.



When you click Next, you'll be asked to name your project and provide a few other fields. These are important, and they are the same in almost all the projects in this book, so please take the time to enter them carefully:

- For Project Name enter "Project1".
- For Team you should select your App Store team if you have one, or use None.
- For Organization Name just use the default value.
- For Organization Identifier enter a unique identifier your app, using reverse domain name format. For example, "com.hackingwithswift.project1" is fine.
- Choose Swift for your language, otherwise the rest of this book will be very confusing indeed.
- Uncheck all three checkboxes.



When you click Next you'll be asked where you want to save the project – somewhere like your desktop is fine. Click Create to finish, and you're done!

Designing a menu

The default Xcode template gives us five interesting files to work with, listed on the left of the Xcode window in the project navigator.

- AppDelegate.swift is there to contain code belonging to your whole application. For example, code that runs when the app is first launched, or when it's being exited.
- ViewController.swift contains code the default screen given to us by the tvOS template. It's almost empty right now because the default screen does nothing right now.
- Main.storyboard contains the user interface design for our app. We'll be using that soon to draw out something more useful.
- Assets.xcassets is an *asset catalog*, which is where you store images to be used in your apps. When Xcode builds your code, asset catalogs get converted into an optimized format for faster loading.
- Info.plist customizes some global app settings, such as its name and default theme.

Before we dive into any of those, I'd like you to press Cmd+R on your keyboard to build and run your code. Alternatively, you can press the Play button that is near the top-left of the Xcode window.

After a second or two, Xcode will finish building your code, and it should launch the tvOS simulator so you can see the app in action. And by "action" I mean completely blank, doing nothing at all – but that's OK, because we only just started.

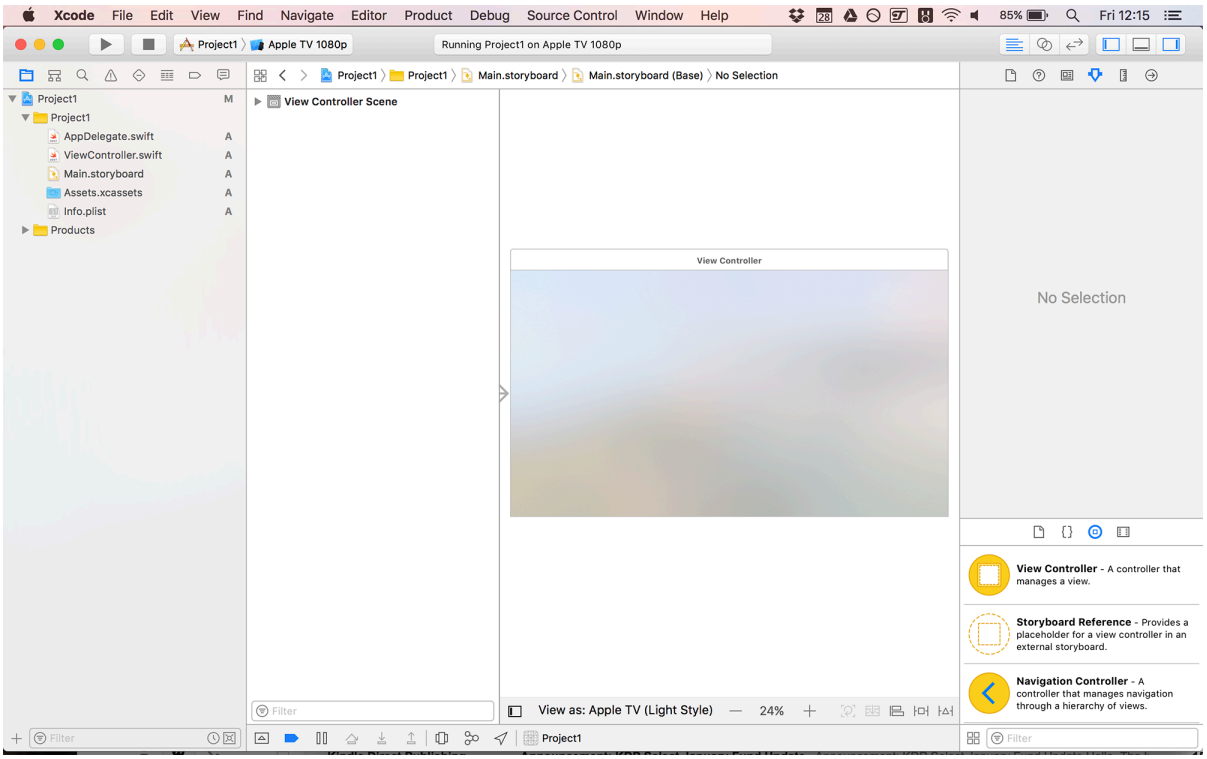
While you're in the simulator, you can leave your app by pressing the Escape key on your keyboard. This is the equivalent of pressing the Menu button on the Siri remote, which should either return to the previous screen or quit the app entirely if there was no previous screen. If you want to quit the app entirely, go back to Xcode and press Cmd+. (a period), which terminates debugging.

OK, let's make our app do something more interesting: open Main.storyboard by selecting it in the project navigator on the left of your Xcode window. This will cause Xcode to load our user interface for editing using Interface Builder – an Xcode component that lets us create

interfaces by dragging and dropping components visually.

Tip: Interface Builder is usually just called IB.

Storyboards let us design multiple screens of our app side by side, much like you might imagine the storyboard of a movie – you can see scenes alongside each other, as well as how the flow moves between them. Right now that's just a single empty screen, so you should see an empty rectangle with the title "View Controller" above it.



Note: if you look at my screenshot you'll see it's made up of four vertical columns. Sometimes Xcode hides the second column, known as the document outline, which is a shame because it's really helpful – if you don't see it, go to the Editor menu and choose Show Document Outline before continuing.

This first screen – a view controller, in Apple parlance – needs three user interface components to make it complete: a logo for our app, a brief description line giving credit to our data source, and a table view, which is a vertically scrolling list component that the user can use to select items. We're going to let the user choose from a list of photo categories they want to view, and

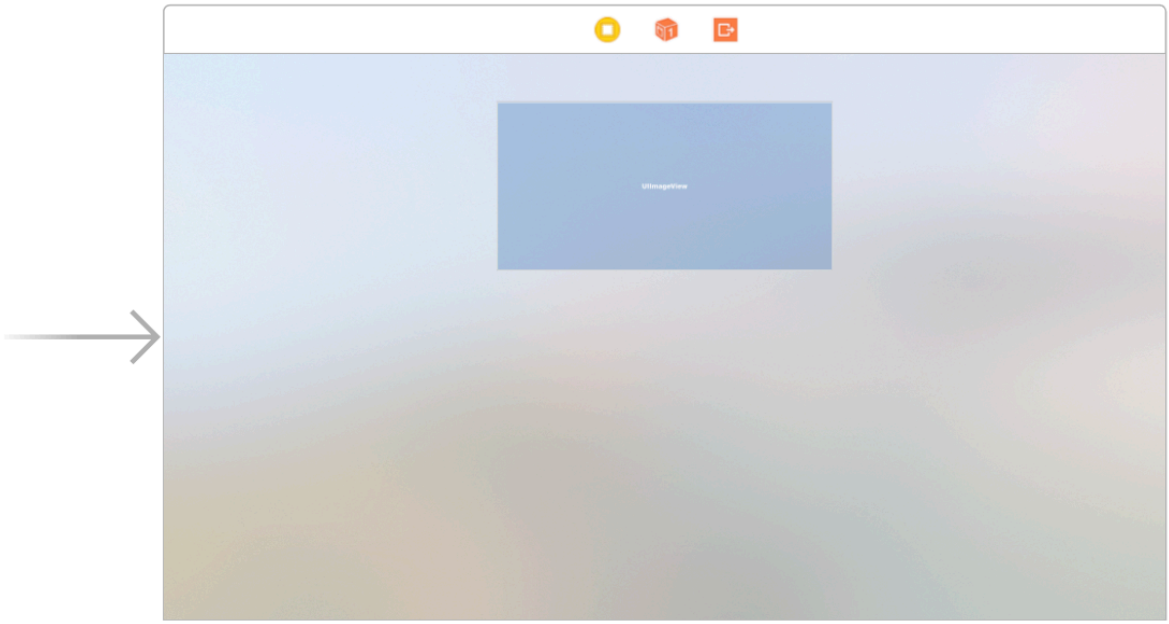
a table view is perfect for that.

If you press Cmd+Shift+L, Xcode should show you the object library – a list of components you can drag out onto the canvas. It should start with "View Controller" at the top, but it's a long list of alternatives.

Tip: At the top of the object library is a button to toggle its layout between grid and table, plus a filter box to let you find things faster. If you currently have the object library in grid mode, I suggest you change to table mode and leave it there, because you get a few lines of description next to each UI component.

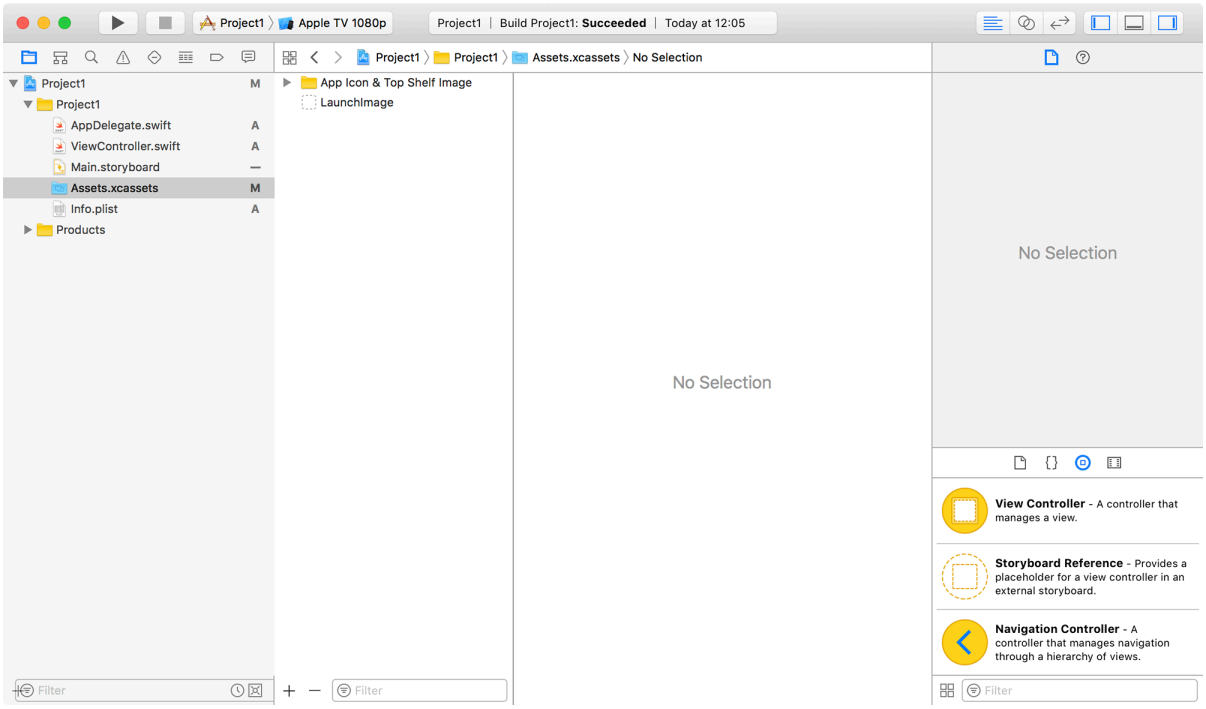
To start with, find "Image View" in the object library, then drag it onto the view controller canvas. You can use the drag handles to resize it if you wish, but it's often easier to type them in directly.

To enter sizes for this image view, go to the size inspector. IB's inspectors are in the top-right of the Xcode window and there are six of them. The size inspector is the last but one, and has a small ruler icon, but it's easiest to bring it up by pressing Alt+Cmd+5 on your keyboard. Give the image an X value of 638, a Y value of 91, then a width of 644 and a height of 323.

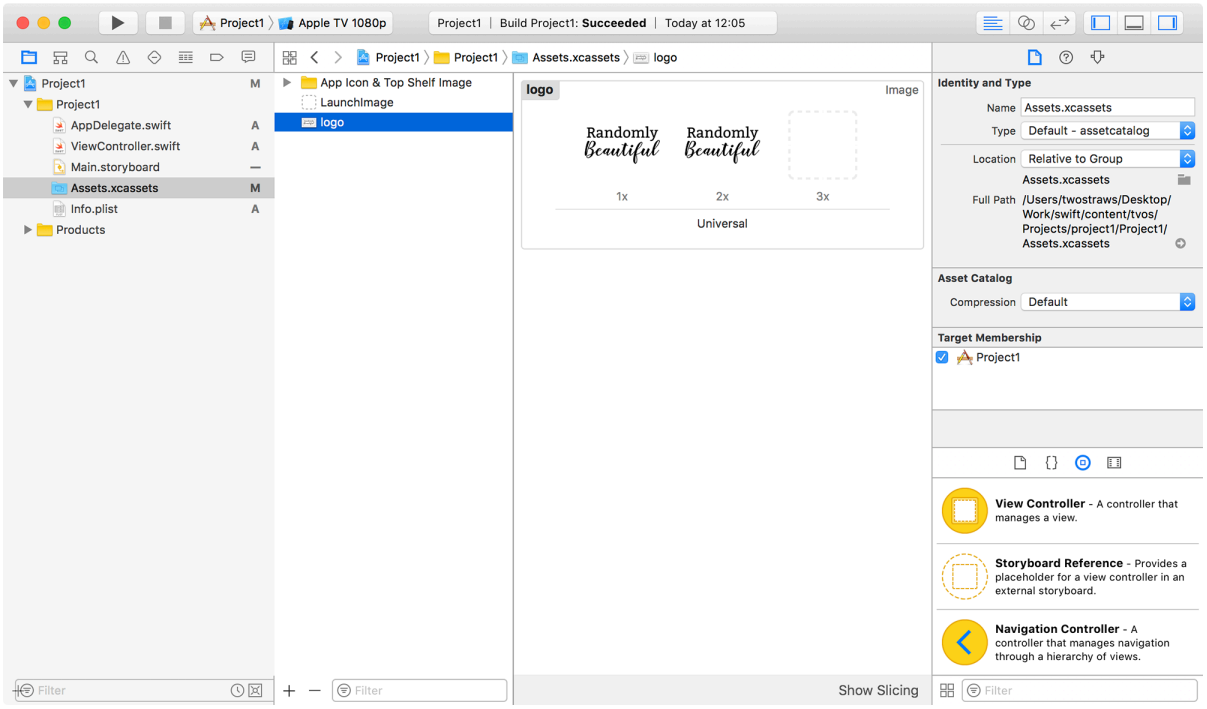


That's a big space for our image, but remember: this needs to be clearly visible across the room. As you might imagine, image views display *images*, so we need to add an image to our project. I already made one for you, and you should have received it with the project files you downloaded with this book. Look inside the project1-files folder and you should see the files "logo.png" and "logo@2x.png" – that's the logo we're going to drop into our new image view.

In Xcode, look for Assets.xcassets in the project navigator. I already said this is where you store image assets, and it's time to use it – please select it now. You should see the IB document outline replaced with two list items: "App Icon & Top Shelf Image", and "LaunchImage".



What you need to do is select logo.png and logo@2x.png in Finder, then drag them into your Xcode asset catalog, into the white space directly below "LaunchImage". When you release your mouse button, "logo" will appear in the list, and next to it you should see the logo with 1x and 2x written by them.



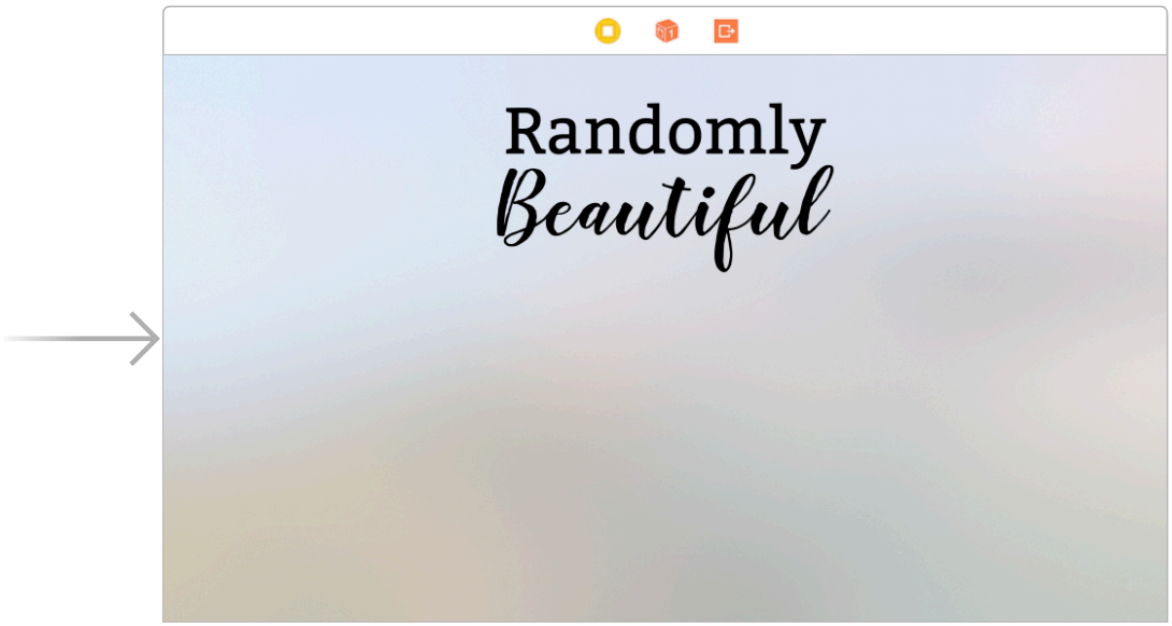
Now, before we continue I want to make an important point: the reason there are two sets of graphics is because I've given you the logo for 1080p TV (logo@1x.png) and the logo for 4K TVs (logo@2x.png). 4K TVs have precisely twice the vertical and horizontal resolution as 1080p TVs, hence the "2x" name.

When you write apps for Apple TV, you ship both sets of assets in your code – one set for regular Apple TVs, and another for Apple TV 4K. When those same apps get *distributed* to users through the App Store, they only ever see *one* set of assets, which is the set that matches their hardware. This means you get to support all devices as intended, but you *don't* end up doubling your app's install size.

Right, back to what actually exists today: we just added our image to the asset catalog, which means we're ready to use it. Go back to Main.storyboard and select the image view, but this time I'd like you to select the *attributes* inspector – it's the one directly to the left of the size inspector, accessible using Alt+Cmd+4.

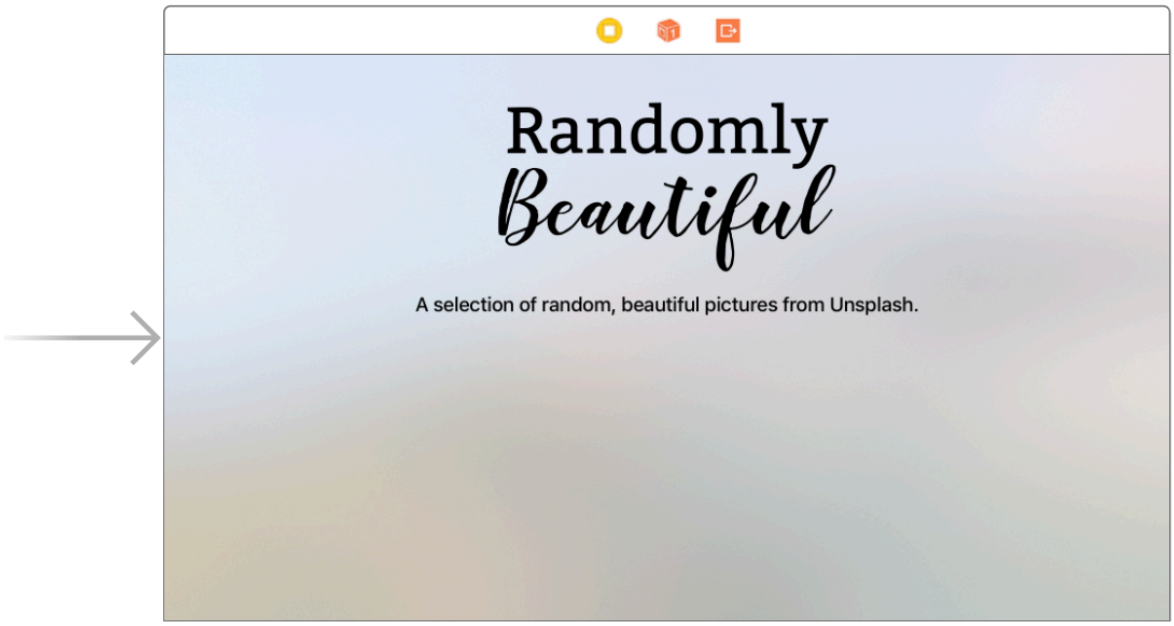
Unlike the relatively simple size inspector, the attributes inspector has a busy job: it contains a huge variety of properties that affect the way UI components look and work, and you'll usually

need to scroll down to see them all. In our case we need the very first option: “Image”. Click the small down arrow to the right of “Image” and choose “logo” from the list – Xcode already scanned our asset catalog to find the image.



Below the image we’re going to write one line of text giving credit to where our images come from. In this app, we’re going to use a website called Unsplash (**https://unsplash.com/**): you have to register in order to use their API, but it’s free to use and the images are all available completely free – free to download, free to modify, free to distribute, free to sell, and so on. However, if you use their API they do ask you give credit to Unsplash and each individual photographer inside your app.

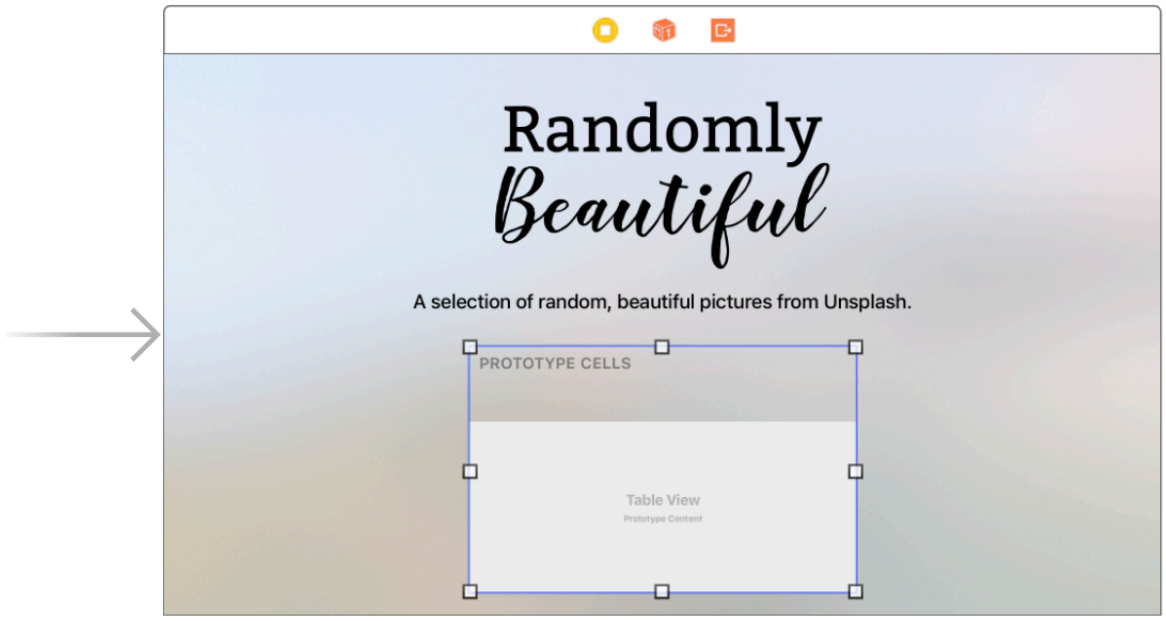
Look in the object library for a Label control, then drag one below the image. Using the size inspector, give it X:445, Y:454, width: 1030, and height: 46. In the attributes inspector, change its alignment to center, then give it this text: “A selection of random, beautiful pictures from Unsplash.”



The last thing we need for this view controller is a table, which is a vertically scrolling list of options the user can choose from. If you search for "table" in the object library you'll see three similar options, but you want the one marked "Table View".

Tip for iOS developers: Table view controllers are very common on iOS, but less so on tvOS because they occupy the full width of the screen – this doesn't look great with such a wide canvas.

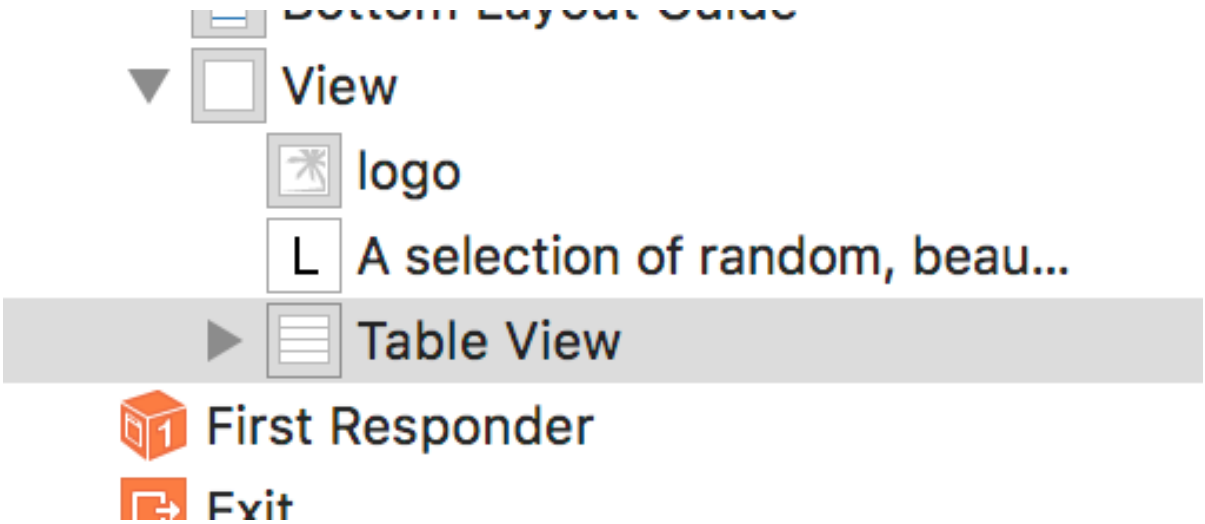
Drag a table view onto the canvas, and use the size inspector to give it the size X:587, Y:562, width: 747, and height: 476.



By default, a table view just a large gray space on your IB canvas, but we're going to fill it with what's called a prototype cell. This is IB's way of designing what our cells ought to look like in an abstract way: what UI components it should contain, where they should be, and so on. Xcode will then create one of these prototype cells for each *real* cell in the table, where it gets injected with real values.

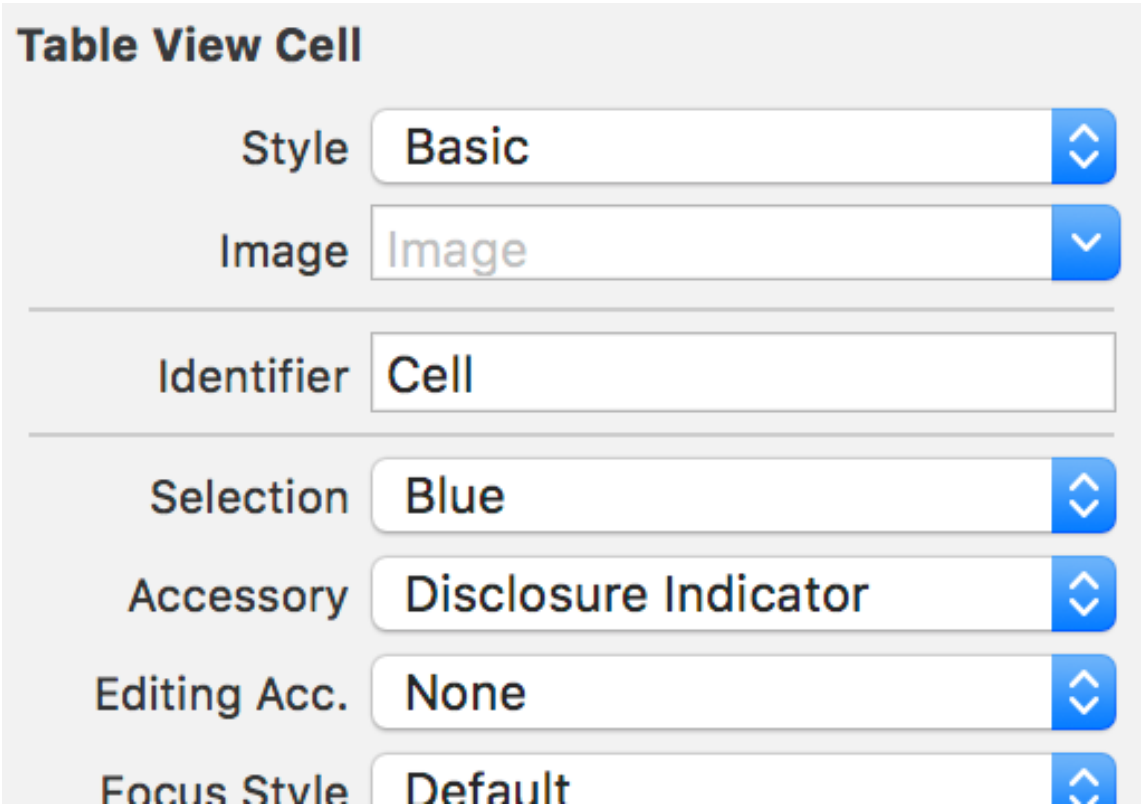
This is an easy app, so we're going to use an easy cell. First, make sure the table view is selected then go to the attributes inspector. Near the top you should see "Prototype Cells: 0" – please change that to 1. You'll see the canvas table view change to show a darker space at the top – that's our prototype cell.

tvOS has four different built-in cell types that we can use without having to do any work ourselves, and we're going to use one of them in this project. To do that, look in the document outline for "Table View" – it should have a small gray disclosure arrow next to it.



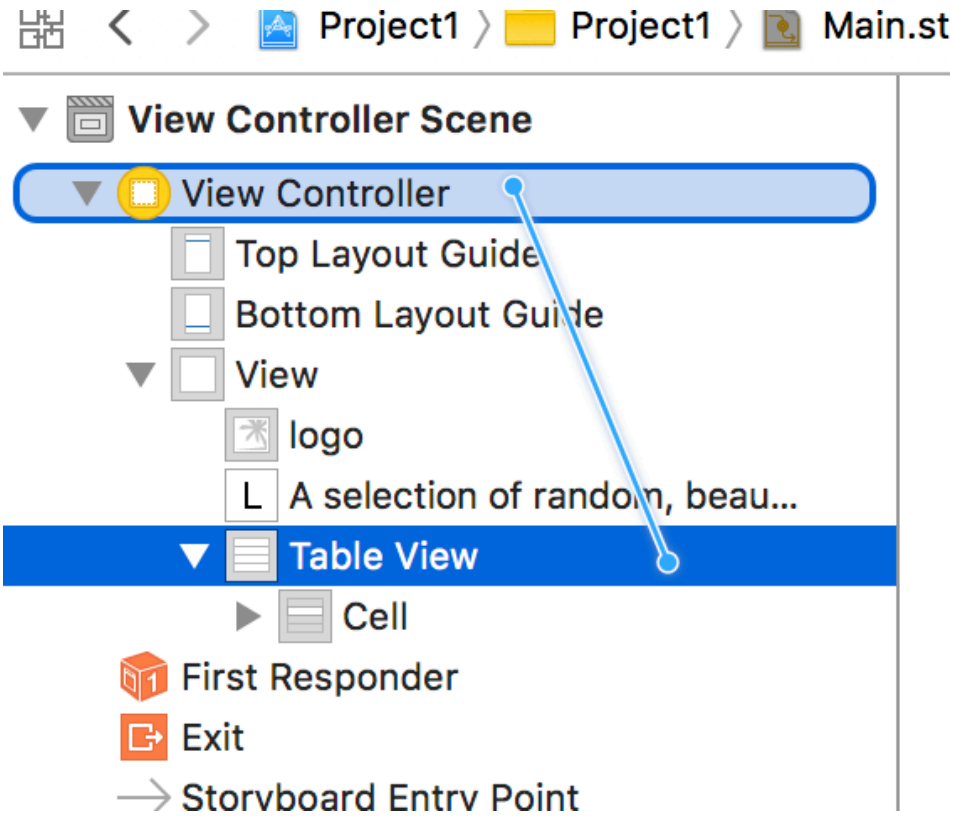
If you open that disclosure arrow now, you'll see "Table View Cell" inside – that's our prototype cell, so please select it. In the attributes inspector you should now see an option saying "Style: Custom". Please change *that* from Custom to Basic, which is a built-in cell prototype that adds one line of large text.

While you're there I'd like you to make two more small changes to this cell. First, look just a little below for the Identifier field, and enter "Cell" in the text box next to it. Second, change "Accessory: None" from None to Disclosure Indicator – this adds a small gray arrow to the right end of the cell.



Before we're done with the user interface for this view controller, we need to tell the table view two things: where it gets its data from, and who should be notified when the user interacts with the table. In both the case the answer is "the view controller that own it," but we still need to tell Xcode that.

In IB, this is done by making a connection between the table view and its view controller. First, select the table view – I suggest you use the document outline for this, because it's possible to click on the wrong thing on the canvas. Now hold down the Ctrl key on your keyboard, and click and drag from the table view up to "View Controller" above it – it has a yellow and white icon next to it.



When you release your mouse button you'll see a menu with two options: "dataSource" and "delegate". Please choose "dataSource", then repeat the procedure and choose "delegate" the second time around.

That's all we need for this view controller, so try pressing Cmd+R to build and run your app. After thinking for a few seconds, the app should install into the simulator and run – only to crash a split second later.

Xcode will throw you back to AppDelegate.swift and highlight a completely meaningless line in red. It will also print a wall of mostly meaningless nonsense into its log, which is an area below the source code editor that appears as needed. If you scroll all the way past the nonsense in the log, you'll finally reach this text: "-[Project1.ViewController tableView: numberOfRowsInSection:]: unrecognized selector sent to instance". Trust me: that the *least* nonsensical part of the error.

Translated, what it's trying to say is "the table view asked its data source (which we set to be

its view controller) how many rows are in the table, and the view controller didn't know what to do." That's fair enough: we did tell the table view to use its view controller for its data source, but we didn't tell the view controller how to do any of that.

To fix this we need to write our first Swift code, inside ViewController.swift. Now, this is a bit confusing so listen carefully: in tvOS a view controller represents one screen of information, and of course screens of information come in an infinite variety of layouts. When we created our app from the Single View App template, Xcode automatically created a new custom view controller for us, which is called **ViewController**.

So, view controllers generally are there to provide any sort of screen we need, but this particular view controller – the one in ViewController.swift that Xcode made for us – is just one type of view controller, and will be used to control the screen we just designed.

Click on ViewController.swift to open it for editing and you should see this code:

```
import UIKit

class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view,
typically from a nib.
    }
}
```

There are five interesting things going on in there:

1. The file starts **import UIKit**, which means "bring in all the functionality from UIKit so I can use it," which means we get all of Apple's user interface toolkit to use.
2. It creates a new custom data type – a class – called **ViewController**. The **: UIViewController** part means "make this build on the existing **UIViewController** class, which is Apple's built-in data type for handling view controllers.

3. The **override func viewDidLoad()** line starts a new method. All the code inside the opening and closing braces ({ and }) form the method body. This method is called by tvOS as soon as our user interface has finished loading – we can put code in here to customize it. It's marked **override** because this method already exists on **UIViewController**, so we're saying "we want our new class to override the one it inherited from its parent."
4. The **super.viewDidLoad()** line means, "tell the class we inherited from to run its own code for this method." This is important: we let **UIViewController** do its thing first, then add our own code after.
5. Finally, all lines that start with // are comments. These are ignored by Xcode, so you can write whatever you want in there. It's a good idea to use comments to document what your code is trying to do.

Tip: When you see a data type that starts with "UI", such as "UIViewController", it means it comes from UIKit.

The two methods Xcode provide for us do nothing other than call their respective methods on **UIViewController**. In Swift, any method you *don't* override will automatically get called on its parent, so we can actually delete these two and nothing will change.

So, change your ViewController.swift file this:

```
import UIKit

class ViewController: UIViewController {

}
```

In IB we made connections between the table view and its view controller to mark the latter as being the data source and delegate for the table. Even though IB is aware of that, Swift isn't – we need to make it clear that our view controller has both those roles.

In Swift, this is done using *protocols*, which are bit like promises. By telling Swift a class conforms to a protocol, you're promising it implements all the methods required to make that

protocol work. In our case we need to make our **ViewController** class conform to two protocols: **UITableViewDataSource** so it says it knows how to serve data to a table, and **UITableViewDelegate** so it says it knows how to respond to user interaction with the table.

Conforming to a protocol is done in two steps. First, you modify the class definition to list the protocols you want to conform to. Edit the **class ViewController** line to this:

```
class ViewController: UIViewController, UITableViewDataSource,
UITableViewDelegate {
```

That now means "create a new data type called "ViewController", that builds on everything already defined by Apple's **UIViewController** class, and specify that this new class conforms to the **UITableViewDataSource** and **UITableViewDelegate** protocols."

The second step in conforming to protocols is to implement the methods those protocols need – to fulfill the promise, as it were. In fact, Xcode will almost certainly have placed a red error symbol next to "class ViewController", because it has detected that we haven't implemented the required methods from those protocols.

To make the error go away, we need to implement two methods: one telling the table view how many rows it should have, and one telling it what should be in each row. As I said already, the table view is there to let users choose a picture category to view, so the first step is to define the categories we're going to show.

Add this code inside the **ViewController** class, directly before the closing brace:

```
var categories = ["Airplanes", "Beaches", "Bridges", "Cats",
"Cities", "Dogs", "Earth", "Forests", "Galaxies", "Landmarks",
"Mountains", "People", "Roads", "Sports", "Sunsets"]
```

That's a *property*, which is a variable that belongs to this class – we can use it in all our methods if we want. You're welcome to change the categories to whatever interest you, but those are fine to get us started.

Hit return a couple of times below that property in order to make some space, then type "numberofrows" – Xcode's code completion system should spring to life and suggest a method called **numberOfRowsInSection**. Press return to accept its suggestion, and you'll see this empty method appear:

```
func tableView(_ tableView: UITableView, numberOfRowsInSection
section: Int) -> Int {

}
```

This is the table view data source method that gets called to figure out how many items you want in the table. We just made the **categories** array containing all the categories we want, so our answer to this is clear: we want as many rows as we have categories.

So, add this line inside the **numberOfRowsInSection** method:

```
return categories.count
```

That's one of the two required methods complete. The next one is a little longer: its job is to create a table view cell using the configuration we made earlier, fill in its text label using the text of the current category, then send it back.

Again, the best way to get started with this is by using code completion, so go to the end of the **numberOfRowsInSection** method – after its closing brace – and press return a few times. Now type "cellfor" and Xcode should offer to autocomplete the **cellForRowAt** method, like this:

```
func tableView(_ tableView: UITableView, cellForRowAt
indexPath: IndexPath) -> UITableViewCell {

}
```

This method, **cellForRowAt**, is called multiple times when your app starts, once for each row

that's visible in the table. As your user scrolls up and down, it gets called again and again so you can can create and configure more rows – it only ever loads just enough for the user to see.

As you might imagine, scrolling a table quickly would mean creating and destroying an awful lot of table view cells, which is a waste of time. To work around this, tvOS lets us reuse cells that would otherwise have been destroyed, which means it effectively reuses the same eight or so cells as the table scrolls up or down.

Let's fill in this method now. First we need to either create a new table view cell or reuse one if possible, and in tvOS that's a single method call: **dequeueReusableCell(withIdentifier:)**. The identifier string is something we set in our storyboard – we named ours "Cell", which is a common default. This method also wants to know the index path – position in the table – of the cell you want to load, because it might already have that cell in its cache. Fortunately, we get passed precisely that as a parameter to the **cellForRowAt** method, so we'll just pass it along.

Add this line of code to the **cellForRowAt** method:

```
let cell = tableView.dequeueReusableCell(withIdentifier:
"Cell", for: indexPath)
```

Next we need to configure the text label of our cell so that it contains the correct category text. When we first created the table view cell it didn't have any text inside, but when we changed its type from Custom to Basic a label appeared saying "Title". This is important: some table cells have text labels and others don't, so even though Xcode will give us back a cell object that has a **textLabel** property we can use, we need to do so carefully because it might have nothing in there.

This is where Swift's optionals come in. Optionals are a way of saying "this might have a value or it might not; don't let me use it directly." Because if you *do* use an empty value directly your app crashes, and no one wants that.

Optionals in Swift use question marks to denote that we're unsure about them. When we do that, Swift adds a lot of boilerplate code behind the scenes: it figures out whether the optional has a value, and if so uses it. If it *doesn't* have a value, Swift stops running this line of code

and moves on to the next one. This means the code works as intended if our optional has a value, but exits safely otherwise.

For example, you could write this:

```
cell.textLabel?.text = "Hello, world!"
```

If **textLabel** doesn't exist – i.e., it's value is set to nil – then the rest of the line does nothing. Perfect!

Back to our method, we need to look up the correct title to put into each row. We can use the **indexPath** parameter for this too, because it tell us the position of the cell being requested. An index path is made up of a section number and a row number, although in this app we're not using sections. So, we can figure out which category to show by looking it up directly like this: **categories[indexPath.row]**.

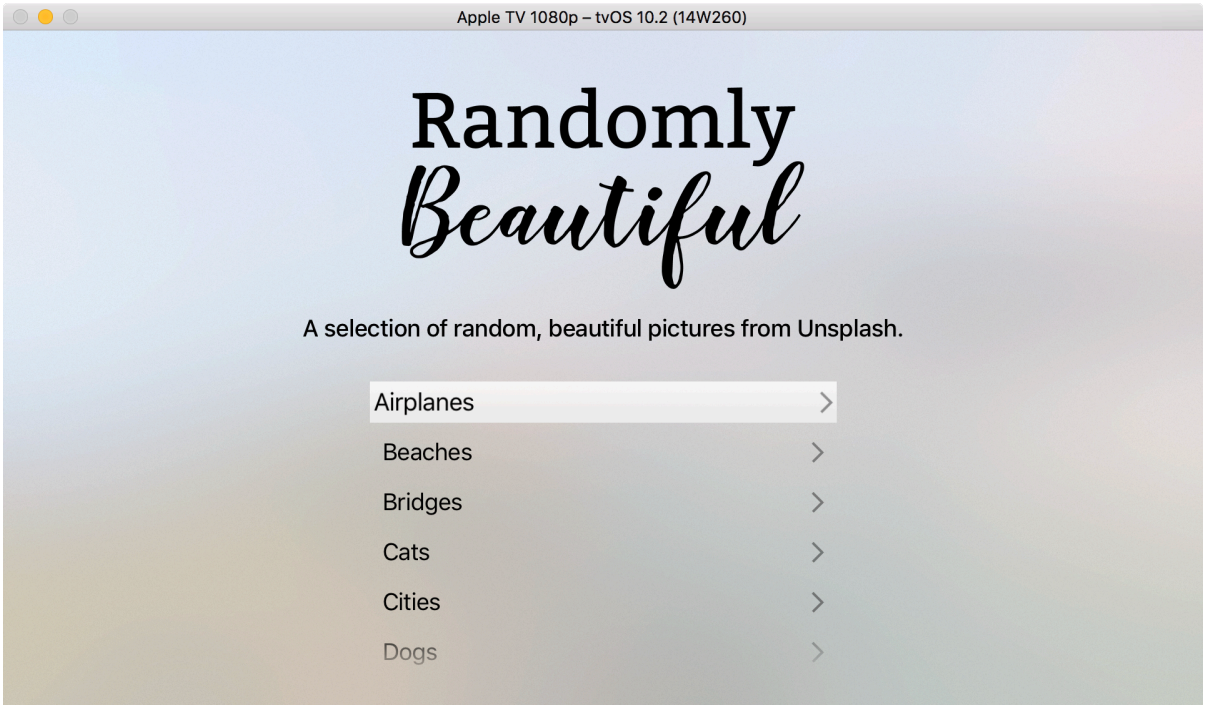
Putting those two pieces of code together, we can write the second line of **cellForRowAt**. Add this code now:

```
cell.textLabel?.text = categories[indexPath.row]
```

Now that we have created and configured the cell we need to send it back so it can be drawn on the screen. To do that, we'll use the **return** keyword to send a value back from the method, passing it our **cell** variable. Add this final line to the **cellForRowAt** method now:

```
return cell
```

With that line of code in place, Xcode's error messages will finally go away because our class conforms fully to the two protocols we requested. In fact, at this point we're finally in a position where we can run the code again, so press Cmd+R to build and run now.

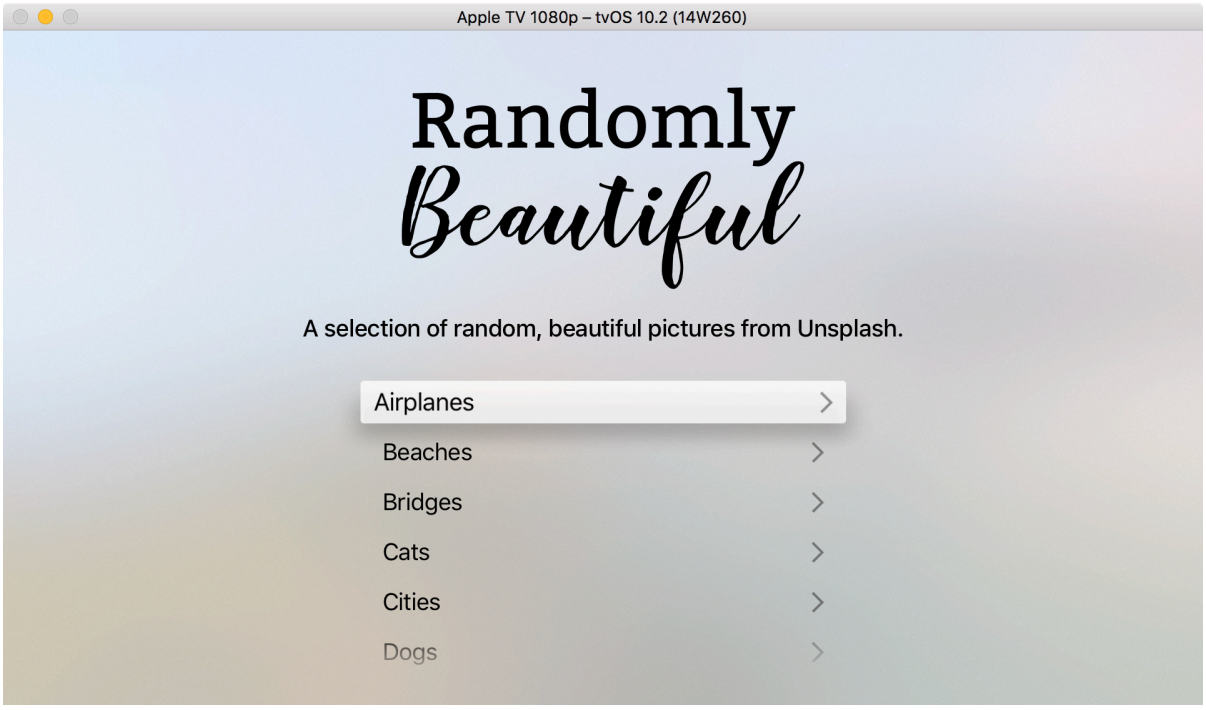


Much better – it doesn't crash any more! You should be able to use the cursor keys on your keyboard to swipe up and down through the rows, or use the touchpad on your Siri remote if you have one connected.

As you move between table rows, you'll notice a gentle animation that highlights what part of the user interface has control right now. In tvOS this is called *focus*, and it's there to help guide users when they are sitting far from the screen.

The effect is a bit subtle by default, because for some unknown reasons tvOS table views are misconfigured by default. If you want to reveal focus in all its glory, re-open Main.storyboard, select the table view cell (it's just called Cell in the document outline), then uncheck its Clips To Bounds box in the attributes inspector.

When you run the program now, you'll see the focus animation effect is more pronounced – cells seem to lift off the rest of the table in 3D, with a gentle shadow behind them.



That's our first screen complete. It's simple but clear, which is a real hallmark of good tvOS design – well done!