Up to date for iOS 10, Xcode 8 & Swift 3

iOS Apprentice

FIFTH EDITION

Tutorial 4: StoreSearch

By Matthijs Hollemans

iOS Apprentice, Fifth Edition

Matthijs Hollemans

Notice of Rights

Notice of Liability

Trademarks

License

About the author

Matthijs Hollemans is a mystic who lives at the top of a mountain where he spends all of his days and nights coding up awesome apps. Actually he lives below sea level in the Netherlands and is pretty down-to-earth but he does spend too much time in Xcode. Check out his website at www.matthijshollemans.com.

About the cover

Striped dolphins live to about 55-60 years of age, can travel in pods numbering in the thousands and can dive to depths of 700 m to feed on fish, cephalopods and crustaceans. Baby dolphins don't sleep for a full a month after they're born. That puts two or three sleepless nights spent debugging code into perspective, doesn't it? :]

Table of Contents: Overview

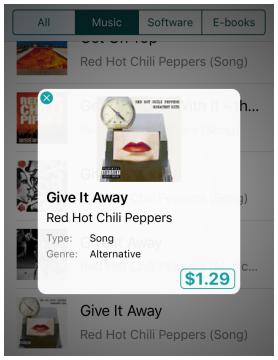
Table of Contents: Extended

Bonus Content

By Matthijs Hollemans

The Detail pop-up

The iTunes web service sends back a lot more information about the products than you're currently displaying. Let's add a "details" screen to the app that pops up when the user taps a row in the table.



The app shows a pop-up when you tap a search result

The table and search bar are still visible in the background, but they have been darkened.

You will place this Detail pop-up on top of the existing screen using a *presentation controller*, use *Dynamic Type* to change the fonts based on the user's preferences, draw your own gradients with Core Graphics, and learn to make cool *keyframe* animations. Fun times ahead!

The to-do list for this section is:

- Design the Detail screen in the storyboard.
- Show this screen when the user taps on a row in the table.

- Put the data from the `SearchResult` into the screen. This includes the item's price, formatted in the proper currency.
- Make the Detail screen appear with a cool animation.

A new screen means a new view controller, so let's start with that.

➤ Add a new **Cocoa Touch Class** file to the project. Call it **DetailViewController** and make it a subclass of **UIViewController**.

You're first going to do the absolute minimum to show this new screen and to dismiss it. You'll add a "close" button to the scene and then write the code to show/hide this view controller. Once that works you will put in the rest of the controls.

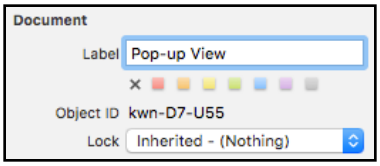
➤ Open the storyboard and drag a new **View Controller** into the canvas. Change its **Class** to **DetailViewController**.

➤ Set the **Background** color of the main view to black, 50% opaque. That makes it easier to see what is going on in the next steps.

➤ Drag a new **View** into the scene. Using the **Size inspector**, make it 240 points wide and 240 high. Center the view in the scene.

➤ In the **Attributes inspector**, change the **Background** color of this new view to white, 95% opaque. This makes it appear slightly translucent, just like navigation bars.

➤ With this new view still selected, go to the **Identity inspector**. In the field where it says "Xcode Specific Label", type **Pop-up View**. You can use this field to give your views names, so they are easier to distinguish inside Interface Builder.



Giving the view a description for use in Xcode

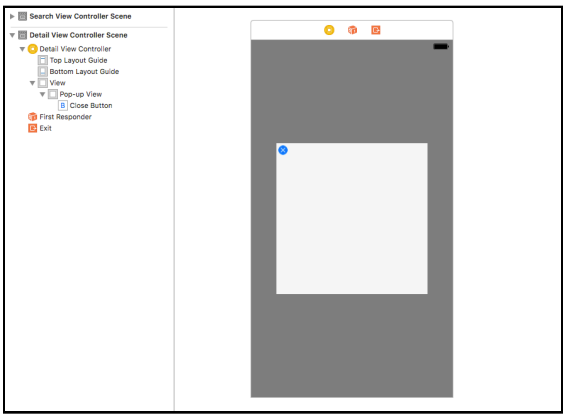
➤ Drag a **Button** into the scene and place it somewhere on the Pop-up View. In the **Attributes** inspector, change **Image** to **CloseButton** (you already added this image to the asset catalog earlier).

➤ Remove the button's text. Choose **Editor → Size to Fit Content** to resize the button and place it in the top-left corner of the Pop-up View (at X = 3 and Y = 0).

➤ If the button's **Type** now says **Custom**, change it back to **System**. That will make the image turn blue (because the default tint color is blue).

➤ Set the Xcode Specific Label for the Button to **Close Button**. Remember that this only changes what the button is called inside Interface Builder; the user will never see that text.

The design should look as follows:



The Detail screen has a white square and a close button on a dark background

> **Note:** Xcode currently gives a warning that this new view controller is unreachable. This warning will disappear after you make a segue to it, which you'll do in a second.

Let's write the code to show and hide this new screen.

➤ In **DetailViewController.swift**, add the following action method:

```
@IBAction func close() {
  dismiss(animated: true, completion: nil)
}
```

There is no need to create a delegate protocol because there's nothing to communicate back to the `SearchViewController`.

➤ Connect this action method to the **X** button's Touch Up Inside event in Interface Builder. (As before, Ctrl-drag from the button to the view controller and pick from Sent Events.)

➤ Ctrl-drag from Search View Controller to Detail View Controller to make a **Present Modally** segue. Give it the identifier **ShowDetail**.

Because the table view doesn't use prototype cells you have to put the segue on the view controller itself. That means you need to trigger the segue manually when the user taps a row.

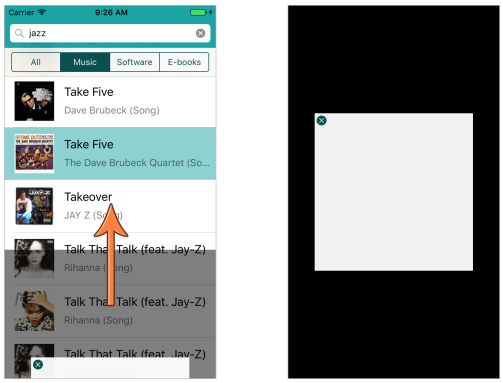
➤ Open **SearchViewController.swift** and change "`didSelectRowAt`" to the following:

```
func tableView(_ tableView: UITableView,
               didSelectRowAt indexPath: IndexPath) {
  tableView.deselectRow(at: indexPath, animated: true)
  performSegue(withIdentifier: "ShowDetail", sender: indexPath)
}
```

You're sending along the index-path of the selected row as the `sender` parameter. This will come in useful later when you're putting the `SearchResult` object into the Detail pop-up. Let's see how well this works.

➤ Run the app and tap on a search result. Hmm, that doesn't look too good.

Even though you set its main view to be half transparent, the Detail screen still has a solid black background. Only during the animation is it see-through:



What happens when you present the Detail screen modally

Hmm, presenting this new screen with a regular modal segue isn't going to achieve the effect we're after.

There are three possible solutions:

1. Don't have a `DetailViewController`. You can load the view for the detail pop-up from a nib and add it as a subview of `SearchViewController`, and put all the logic for this screen in `SearchViewController` as well. This is not a very good solution because it makes `SearchViewController` more complex – the logic for a new screen should really go into its own view controller.

2. Use the *view controller containment* APIs to embed the `DetailViewController` "inside" the `SearchViewController`. This is a better solution but still more work than necessary. (You'll see an example of view controller containment in the next section where you'll be adding a special landscape mode to the app.)

3. Use a *presentation controller*. This lets you customize how modal segues present their view controllers on the screen. You can even have custom animations to show and hide the view controllers.

Let's go for #3. Transitioning from one screen to another in an iOS app involves a complex web of objects that take care of all the details concerning presentations, transitions, and animations. Normally, that all happens behind the scenes and you can safely ignore it.

But if you want to customize how some of this works, you'll have to dive into the excitingly strange world of presentation controllers and transitioning delegates.

➤ Add a new Swift File to the project, named **DimmingPresentationController**.

➤ Replace the contents of this new file with the following:

```
import UIKit

class DimmingPresentationController: UIPresentationController {
  override var shouldRemovePresentersView: Bool {
    return false
  }
}
```

The standard `UIPresentationController` class contains all the logic for presenting new view controllers. You're providing your own version that overrides some of this behavior, in particular telling UIKit to leave the `SearchViewController` visible. That's necessary to get the see-through effect.

In a short while you'll also add a light-to-dark gradient background view to this presentation controller; that's where the "dimming" in its name comes from.

> **Note:** It's called a presentation controller, but it is not a *view* controller. The use of the word controller may be a bit confusing here but not all controllers are for managing screens in your app (only those with "view" in their name).

> A presentation controller is an object that "controls" the presentation of something, just like a view controller is an object that controls a view and everything in it. Soon you'll also see an animation controller, which controls – you guessed it – an animation.
>
> There are quite a few different kinds of controller objects in the various iOS frameworks. Just remember that there's a difference between a view controller and other types of controllers.

You need to tell the app you want to use your own presentation controller to show the Detail pop-up.

➤ In **DetailViewController.swift**, add the following extension at the very bottom of the file:

```
extension DetailViewController: UIViewControllerTransitioningDelegate {

  func presentationController(forPresented presented: UIViewController,
                              presenting: UIViewController?,
                              source: UIViewController)
                              -> UIPresentationController? {
    return DimmingPresentationController(
             presentedViewController: presented, presenting: presenting)
  }
}
```

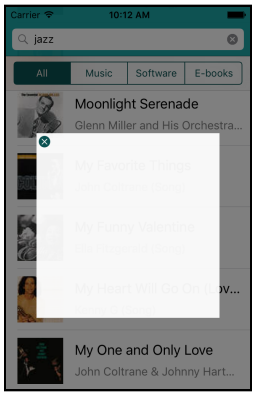
The methods from this delegate protocol tell UIKit what objects it should use to perform the transition to the Detail View Controller. It will now use your new `DimmingPresentationController` class instead of the standard presentation controller.

➤ Also add the `init?(coder)` method to class `DetailViewController`:

```
required init?(coder aDecoder: NSCoder) {
  super.init(coder: aDecoder)
  modalPresentationStyle = .custom
  transitioningDelegate = self
}
```

Recall that `init?(coder)` is invoked to load the view controller from the storyboard. Here you tell UIKit that this view controller uses a custom presentation and you set the delegate that will call the method you just implemented.

➤ Run the app again and tap a row to bring up the detail pop-up. That looks better! Now the list of search results remains visible.



The Detail pop-up background is now see-through

The standard presentation controller removed the underlying view from the screen, making it appear as if the Detail pop-up had a solid black background. Removing the view makes sense most of the time when you present a modal screen, as the

user won't be able to see the previous screen anyway (not having to redraw this view saves battery power too).

However, in our case the modal segue leads to a view controller that only partially covers the previous screen. You want to keep the underlying view to get the see-through effect. That's why you needed to supply your own presentation controller object.

Later on you'll add custom animations to this transition and for that you need to tweak the presentation controller some more and also provide your own animation controller objects.

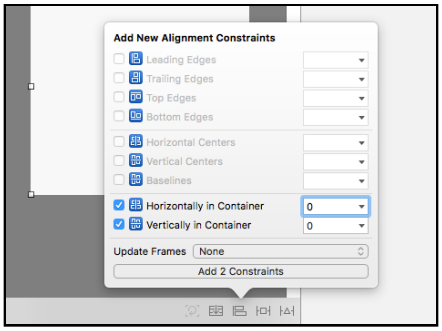
➤ Also verify that the close button works to dismiss the pop-up.

Now run the app on the iPhone 7 Plus simulator. What happens? The Detail pop-up isn't properly centered in the screen anymore.

Exercise. What do you need to do to center the Detail pop-up? ■

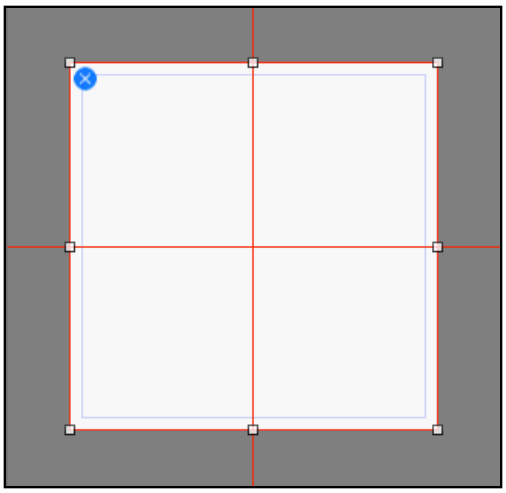
Answer: Add some Auto Layout constraints, of course! The current design of the Detail screen is for the iPhone SE. When the app runs on a larger device, UIKit doesn't know yet that it should keep the pop-up view centered.

➤ In the storyboard, select the **Pop-up View**. Click the **Align** button at the bottom of the canvas and put checkmarks in front of **Horizontally in Container** and **Vertically in Container**.



Adding constraints to align the Pop-up View

➤ Press **Add 2 Constraints** to finish. This adds two new constraints to the Pop-up View that keep it centered, represented by the red lines that cross the scene.



The Pop-up View with alignment constraints

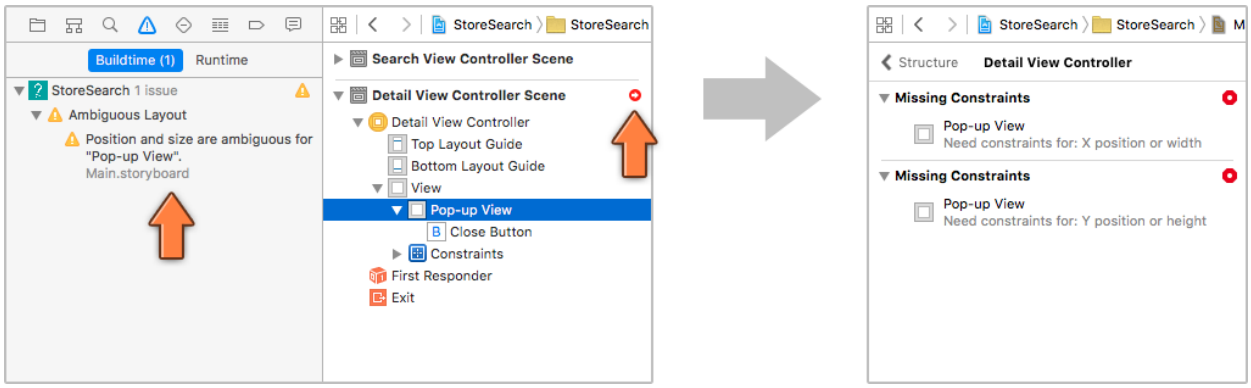
One small hiccup: these lines are supposed to be blue, not red. Whenever you see red or orange lines, Auto Layout has a problem.

The number one rule for using Auto Layout is this: For each view you always need enough constraints to determine both its position and size.

Before you added your own constraints, Xcode gave automatic constraints to the Pop-up View, based on where you placed that view in Interface Builder. But as soon as you add a single constraint of your own, you no longer get these automatic constraints.

The Pop-up View has two constraints that determine the view's position – it is always centered horizontally and vertically in the window – but there are no constraints yet for its size.

Xcode is helpful enough to point this out in the Issue navigator:



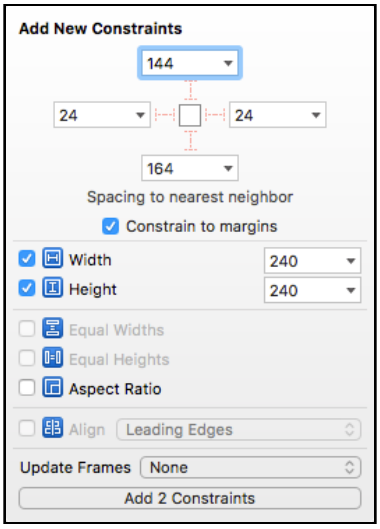
Xcode shows Auto Layout errors in the Issue navigator

➤ Tap the small red arrow in the outline pane to get a more detailed explanation of the errors. It's obvious that something's missing. You know it's not the position –

the two alignment constraints are enough to determine that – so it must be the size.

The easiest way to fix these errors is to give the Pop-up View fixed width and height constraints.

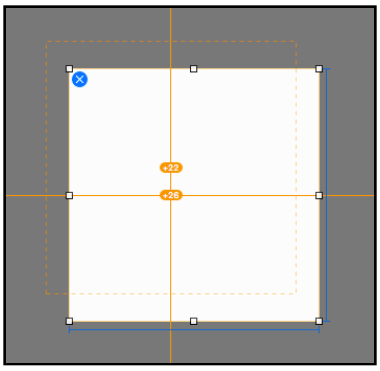
➤ Select the Pop-up View and click the **Add New Constraints** button. Put checkmarks in front of **Width** and **Height**. Click **Add 2 Constraints** to finish.



Pinning the width and height of the Pop-up View

Now the lines turn blue and Auto Layout is happy.

Note: If your lines do not turn blue and the design looks something like the following, then your constraints and the view's frame do not match up.



Auto Layout believes the view is misplaced

In other words, Auto Layout thinks that the Pop-up view should be placed where the orange dotted box is but you've put it somewhere else. To fix this, click the **Update Frames** button at the bottom of the Interface Builder canvas.

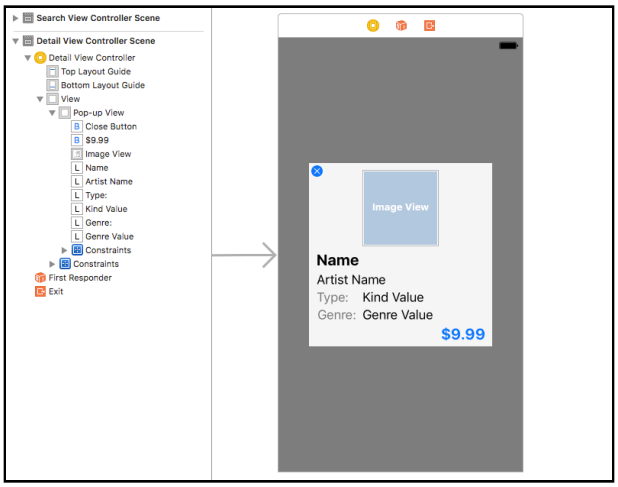
The Update Frames button

You can also choose the menu item **Editor → Resolve Auto Layout Issues → Update Frames** to fix the Pop-up view's position.

➤ Run the app on the different Simulators and verify that the pop-up now always shows up in the exact center of the screen.

Adding the rest of the controls

Let's finish the design of the Detail screen. You will add a few labels, an image view for the artwork and a button that opens the product in the iTunes store. The design will look like this:



The Detail screen with the rest of the controls

➤ Drag a new **Image View**, six **Labels**, and a **Button** into the canvas and build a layout like the one from the picture.

Some suggestions for the dimensions:

Control	X	Y	Width	Height
Image View	70	9	100	100
Name label	10	115	220	24
Artist Name label	10	142	220	21
Type: label	10	165	43	21
Kind Value label	70	165	160	21
Genre: label	10	188	51	21
Genre Value label	70	188	160	21
$9.99 button	168	212	68	24

➤ The **Name** label's font is **System Bold 20**. Set **Autoshrink** to **Minimum Font Scale** so the font can become smaller if necessary to fit as much text as possible.

➤ The font for the **$9.99** button is also **System Bold 20**. In a moment you will also give this button a background image.

➤ You shouldn't have to change the font for the other labels; they use the default value of System 17.

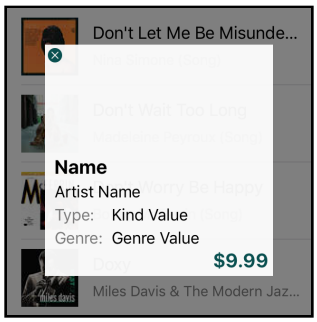
➤ Set the **Color** for the **Type:** and **Genre:** labels to 50% opaque black.

These new controls are pretty useless without outlet properties, so add the following lines to **DetailViewController.swift**:

```
@IBOutlet weak var popupView: UIView!
@IBOutlet weak var artworkImageView: UIImageView!
@IBOutlet weak var nameLabel: UILabel!
@IBOutlet weak var artistNameLabel: UILabel!
@IBOutlet weak var kindLabel: UILabel!
@IBOutlet weak var genreLabel: UILabel!
@IBOutlet weak var priceButton: UIButton!
```

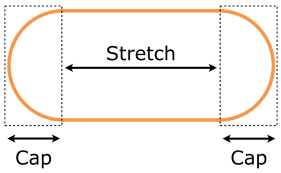
➤ Connect the outlets to the views in the storyboard. Ctrl-drag from Detail View Controller to each of the views and pick the corresponding outlet. (The Type: and Genre: labels and the X button do not get an outlet.)

➤ Run the app to see if everything still works.



The new controls in the Detail pop-up

The reason you did not put a background image on the price button yet is that I want to tell you about **stretchable images**. When you put a background image on a button in Interface Builder, it always has to fit the button exactly. That works fine in many cases, but it's more flexible to use an image that can stretch to any size.

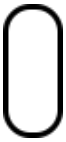


The caps are not stretched but the inner part of the image is

When an image view is wider than the image, it will automatically stretch the image to fit. In the case of a button, however, you don't want to stretch the ends (or "caps") of the button, only the middle part. That's what a stretchable image lets you do.

In the Bull's Eye tutorial you used `resizableImage(withCapInsets)` to cut the images for the slider track into stretchable parts. You can also do this in the asset catalog without having to write any code.

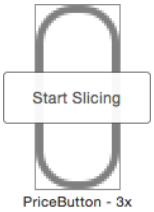
➤ Open **Assets.xcassets** and select the **PriceButton** image set.



The PriceButton image

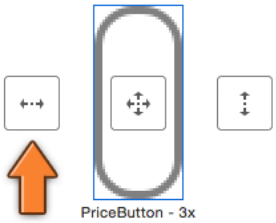
If you take a detailed look at this image you will see that it is only 11 points wide. That means it has a 5-point cap on the left, a 5-point cap on the right, and a 1-point body that will be stretched out.

Click **Show Slicing** at the bottom.



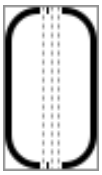
The Start Slicing button

Now all you have to do is click **Start Slicing** on each of the three images, followed by the **Slice Horizontally** button:



The Slice Horizontally button

You should end up with something like this for each of the button sizes:



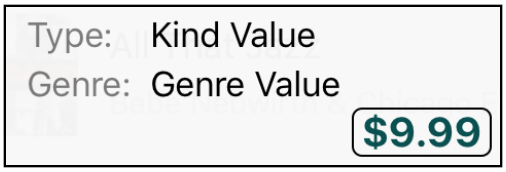
After slicing

Each image is cut into three parts: the caps on the end and a one-pixel area in the middle that is the stretchable part. Now when you put this image onto a button or inside a `UIImageView`, it will automatically stretch itself to whatever size it needs to be.

➤ Go back to the storyboard. For the $9.99 button, change **Background** to **PriceButton**.

If you see the image repeating, make sure that the button is only 24 points high, the same as the image height.

➤ Run the app and check out that button. Here's a close-up of what it looks like:



The price button with the stretchable background image

The main reason you're using a stretchable image here is that the text on the button may vary in size so you don't know in advance how big the button needs to be. If your app has a lot of custom buttons, it's worth making their images stretchable. That way you won't have to re-do the images whenever you're tweaking the sizes of the buttons.

The button could still look a little better, though – a black frame around dark green text doesn't particularly please the eye. You could go into Photoshop and change the color of the image to match the text color, but there's an easier method.

The color of the button text comes from the global tint color. `UIImage` makes it very easy to make images appear in the same tint color.

➤ In the asset catalog, select the **PriceButton** set again and go to the **Attribute inspector**. Change **Render As** to **Template Image**.

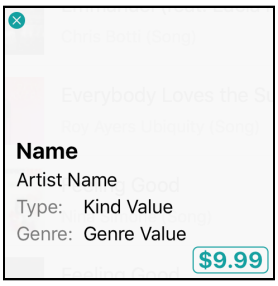
When you set the "template" rendering mode on an image, UIKit removes the original colors from the image and paints the whole thing in the tint color.

I like the dark green tint color in the rest of the app but for this pop-up it's a bit too dark. You can change the tint color on a per-view basis; if that view has subviews the new tint color also applies to these subviews.

➤ In **DetailViewController.swift**, add the line that sets the `tintColor` to `viewDidLoad()`:

```
override func viewDidLoad() {
  super.viewDidLoad()
  view.tintColor = UIColor(red: 20/255, green: 160/255, blue: 160/255,
                           alpha: 1)
}
```

Note that you're setting the new `tintColor` on `view`, not just on `priceButton`. That will apply the lighter tint color to the close button as well.



The buttons appear in the new tint color

Much better, but there is still more to tweak. In the screenshot I showed you at the start of this section, the pop-up view had rounded corners. You could use an image to make it look like that but instead I'll show you a little trick.

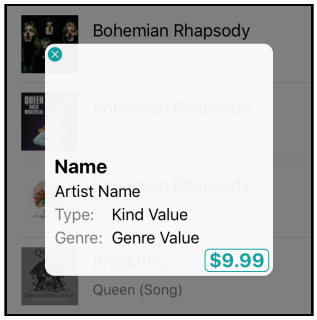
`UIViews` do their drawing using a so-called `CALayer` object. The CA prefix stands for Core Animation, which is the awesome framework that makes animations so easy on the iPhone. You don't need to know much about those "layers", except that each view has one, and that layers have some handy properties.

➤ Add the following line to `viewDidLoad()`:

```
popupView.layer.cornerRadius = 10
```

You ask the Pop-up View for its layer and then set the corner radius of that layer to 10 points. And that's all you need to do!

➤ Run the app. There's your rounded corners:



The pop-up now has rounded corners

The close button is pretty small, about 15 by 15 points. From the Simulator it is easy to click because you're using a precision pointing device (the mouse). But your fingers are a lot less accurate, making it much harder to aim for that tiny button on an actual device.

That's one reason why you should always test your apps on real devices and not just on the Simulator. (Apple recommends that buttons always have a tap area of at least 44×44 points.)

To make the app more user-friendly, you'll also allow users to dismiss the pop-up by tapping anywhere outside it. The ideal tool for this job is a **gesture recognizer**.

➤ Add a new extension to **DetailViewController.swift**:

```
extension DetailViewController: UIGestureRecognizerDelegate {
  func gestureRecognizer(_ gestureRecognizer: UIGestureRecognizer,
                         shouldReceive touch: UITouch) -> Bool {
    return (touch.view === self.view)
  }
}
```

You only want to close the Detail screen when the user taps outside the pop-up, i.e. on the background. Any other taps should be ignored. That's what this delegate method is for. It only returns `true` when the touch was on the background view but `false` if it was inside the Pop-up View.

Note that you're using the identity operator === to compare `touch.view` with `self.view`. You want to know whether both variables refer to the same object. This is different from using the == equality operator. That would check whether both variables refer to objects that are considered equal, even if they aren't the same object. (Using == here would have worked too, but only because `UIView` treats == and === the same. But not all objects do, so be careful!)

➤ Add the following lines to `viewDidLoad()`:

```
let gestureRecognizer = UITapGestureRecognizer(target: self,
                                               action: #selector(close))
gestureRecognizer.cancelsTouchesInView = false
gestureRecognizer.delegate = self
view.addGestureRecognizer(gestureRecognizer)
```

This creates the new gesture recognizer that listens to taps anywhere in this view controller and calls the `close()` method in response.

➤ Try it out. You can now dismiss the pop-up by tapping anywhere outside the white pop-up area. That's a common thing that users expect to be able to do, and it was easy enough to add to the app. Win-win!

Putting the data into the Detail pop-up

Now that the app can show this pop-up after a tap on a search result, you should put the name, genre and price from the selected product in the pop-up.

Exercise. Try to do this by yourself. It's not any different from what you've done in the past tutorials! ■

There is more than one way to pull this off, but I like to do it by putting the `SearchResult` object in a property on the `DetailViewController`.

➤ Add this property to **DetailViewController.swift**:

```
var searchResult: SearchResult!
```

As usual, this is an implicitly-unwrapped optional because you won't know what its value will be until the segue is performed. It is `nil` in the mean time.

➤ Also add a new method, `updateUI()`:

```
func updateUI() {
  nameLabel.text = searchResult.name

  if searchResult.artistName.isEmpty {
    artistNameLabel.text = "Unknown"
  } else {
    artistNameLabel.text = searchResult.artistName
  }

  kindLabel.text = searchResult.kind
  genreLabel.text = searchResult.genre
}
```

That looks very similar to what you did in `SearchResultCell`.

The logic for setting the text on the labels has its own method, `updateUI()`, because that is cleaner than stuffing everything into `viewDidLoad()`.

➤ Call the new method from `viewDidLoad()`:

```
override func viewDidLoad() {
  super.viewDidLoad()
  . . .

  if searchResult != nil {
    updateUI()
  }
}
```

The `if != nil` check is a defensive measure, just in case the developer forgets to fill in `searchResult` on the segue.

(Note: You can also write this as `if let _ = searchResult` to unwrap the optional. Because you're not actually using the unwrapped value for anything, you specify the _ wildcard symbol.)

The Detail pop-up is launched with a segue triggered from `SearchViewController`'s `tableView(didSelectRowAt)`.

You'll have to add a `prepare(for:sender:)` method to configure the `DetailViewController` when the segue happens.

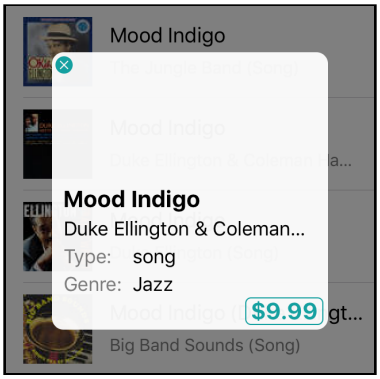
➤ Add this method to **SearchViewController.swift**:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
 if segue.identifier == "ShowDetail" {
   let detailViewController = segue.destination as! DetailViewController
   let indexPath = sender as! IndexPath
   let searchResult = searchResults[indexPath.row]
   detailViewController.searchResult = searchResult
 }
}
```

This should hold no big surprises for you.

When "`didSelectRowAt`" starts the segue, it sends along the index-path of the selected row. That lets you find the `SearchResult` object and put it in `DetailViewController`'s property.

➤ Try it out. All right, that's starting to look like something:



The pop-up with filled-in data

One thing you did in `SearchResultCell` was translating the `kind` value from an internal identifier to something that looks a bit better to humans. That logic, in the form of the `kindForDisplay()` method, sits in `SearchResultCell`, but now I'd like to use it in `DetailViewController` as well. Problem: the `DetailViewController` doesn't have anything to do with `SearchResultCell`.

You could simply copy-paste the `kindForDisplay()` method but then you have identical code in two different places in the app.

What if you decide to support another type of product, then you'd have to remember to update this method in two places as well. That sort of thing becomes a maintenance nightmare and is best avoided. Instead, you should look for a better place to put that method.

Exercise: Where would you put it? ■

Answer: `kind` is a property on the `SearchResult` object. It makes sense that you can also ask the `SearchResult` for a nicer version of that value, so let's move the entire `kindForDisplay()` method into the `SearchResult` class.

➤ Cut the `kindForDisplay()` method out of the `SearchResultCell` source code and put it in **SearchResult.swift**, inside `class SearchResult`.

Because `SearchResult` already has a `kind` property, you don't have to pass that value as a parameter to this method.

➤ Change the method signature to:

```
func kindForDisplay() -> String {
```

Of course, **SearchResultCell.swift**'s `configure(for)` now tries to call a method that no longer exists.

➤ Fix the following line in `configure(for)`:

```
artistNameLabel.text = String(format: "%@ (%@)",
                searchResult.artistName, searchResult.kindForDisplay())
```

Let's also call this new method in **DetailViewController.swift**.

➤ Change the line in `updateUI()` that sets the "kind" label to:

```
kindLabel.text = searchResult.kindForDisplay()
```

Nice, you refactored the code to make it cleaner and more powerful. I often start out by putting all my code in the view controllers but as the app evolves, more and more gets moved into their own classes where it really belongs.

In retrospect, the `kindForDisplay()` method really returns a property of `SearchResult` in a slightly different form, so it is functionality that logically goes with the `SearchResult` object, not with its cell or the view controller.

It's OK to start out with your code being a bit of a mess – that's what it often is for me! – but whenever you see an opportunity to clean things up and simplify it, you should take it.

As your source code evolves, it will become clearer what the best internal structure is for that particular program. But you have to be willing to revise the code when you realize it can be improved in some way!

➤ Run the app. The "Type" label in the pop-up should now have the same polished text as the list of search results.

There are three more things to do on this screen:

1. Show the price, in the proper currency.

2. Make the price button open the product page in the iTunes store.
3. Download and show the artwork image. This image is slightly larger than the one from the table view cell.

These are all fairly small features so you should be able to do them quite quickly. The price goes first.

➤ Add the following code to `updateUI()`:

```
let formatter = NumberFormatter()
formatter.numberStyle = .currency
formatter.currencyCode = searchResult.currency

let priceText: String
if searchResult.price == 0 {
  priceText = "Free"
} else if let text = formatter.string(
                                  from: searchResult.price as NSNumber) {
  priceText = text
} else {
  priceText = ""
}

priceButton.setTitle(priceText, for: .normal)
```

You've used `DateFormatter` in previous tutorials to turn a `Date` object into human-readable text. Here you use `NumberFormatter` to do the same thing for numbers.

In the past tutorials you've turned numbers into text using string interpolation `\(…)` and `String(format:)` with the `%f` or `%d` format specifier. However, in this case you're not dealing with regular numbers but with money in a certain currency.

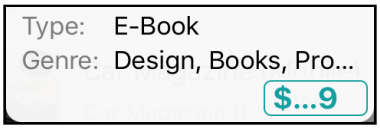
There are different rules for displaying various currencies, especially if you take the user's language and country settings into consideration. You could program all of these rules yourself, which is a lot of effort, or choose to ignore them. Fortunately, you don't have to make that tradeoff because you have `NumberFormatter` to do all the hard work.

You simply tell the `NumberFormatter` that you want to display a currency value and what the currency code is. That currency code comes from the web service and is something like "USD" or "EUR". `NumberFormatter` will insert the proper symbol, such as $ or € or ¥, and formats the monetary amount according to the user's regional settings.

There's one caveat: if you're not feeding `NumberFormatter` an actual number, it cannot do the conversion. That's why `string(from)` returns an optional that you need to unwrap.

➤ Run the app and see if you can find some good deals. :-)

Occasionally you might see this:



The price doesn't fit into the button

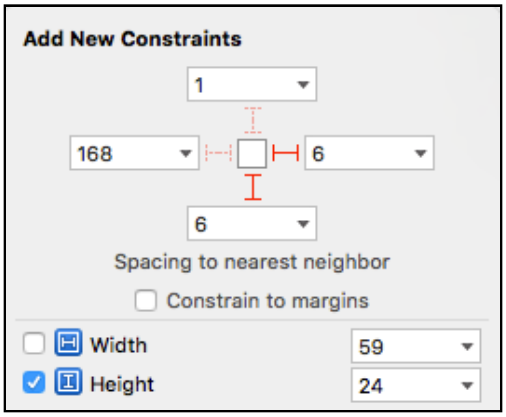
When you designed the storyboard you made this button 68 points wide. You didn't put any constraints on it, so Xcode gave it an automatic constraint that always forces the button to be 68 points wide, no more, no less.

But buttons, like labels, are perfectly able to determine what their ideal size is based on the amount of text they contain. That's called the **intrinsic content size**.

➤ Open the storyboard and select the price button. Choose **Editor → Size to Fit Content** from the menu bar (or press ⌘=). This resizes the button to its ideal size, based on the current text.

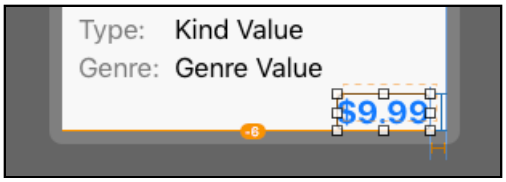
That alone is not enough. You also need to add at least one constraint to the button or Xcode will still apply the automatic constraints.

➤ With the price button selected, click the **Add New Constraints** button. Add two spacing constraints, one on the right and one on the bottom, both 6 points in size. Also add a 24-point Height constraint:



Pinning the price button

Don't worry if your storyboard now looks something like this:



Orange bars indicate the button is misplaced

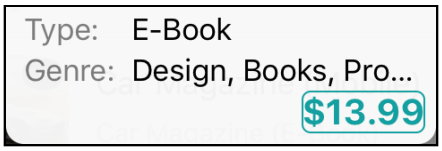
The orange lines simply mean that the current position and/or size of the button in the storyboard does not correspond to the position and size that Auto Layout calculated from the constraints. This is easily fixed:

➤ Select the button and from the menu bar choose **Editor → Resolve Auto Layout Issues → Update Frames**. Now the lines should all turn blue.

To recap, you have set the following constraints on the button:

- Fixed height of 24 points. That is necessary because the background image is 24 points tall.
- Pinned to the right edge of the pop-up with a distance of 6 points. When the button needs to grow to accommodate larger prices, it will extend towards the left. Its right edge always stays aligned with the right edge of the pop-up.
- Pinned to the bottom of the pop-up, also with a distance of 6 points.
- There is no constraint for the width. That means the button will use its intrinsic width – the larger the text, the wider the button. And that's exactly what you want to happen here.

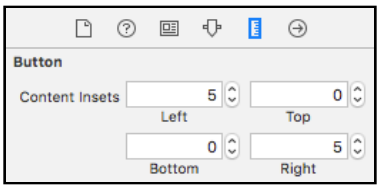
➤ Run the app again and pick an expensive product (something with a price over $9.99; e-books are a good category for this).



The button is a little cramped

That's better but the text now runs into the border from the background image. You can fix this by setting the "content edge insets" for the button.

➤ Go to the **Size inspector** and find where it says **Content Insets**. Change **Left** and **Right** to 5.

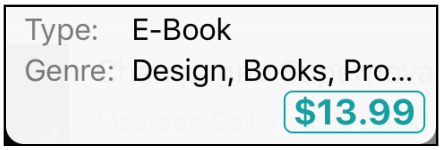


Changing the content edge insets of the button

This adds 5 points of padding on the left and right sides of the button. Of course, this causes the button's frame to be misplaced again because it is now 10 points wider.

➤ Click the **Update Frames** button or choose the **Editor → Resolve Auto Layout Issues → Update Frames** menu item to fix it.

➤ Run the app; now the price button should finally look good:



That price button looks so good you almost want to tap it!

Tapping the button should take the user to the selected product's page on the iTunes store.

➤ Add the following method to **DetailViewController.swift**:

```
@IBAction func openInStore() {
  if let url = URL(string: searchResult.storeURL) {
    UIApplication.shared.open(url, options: [:], completionHandler: nil)
  }
}
```

➤ And connect the `openInStore` action to the button's Touch Up Inside event (in the storyboard).

That's all you have to do. The web service returned a URL to the product page. You simply tell the `UIApplication` object to open this URL. iOS will now figure out what sort of URL it is and launch the proper app in response – iTunes Store, App Store, or Mobile Safari. (On the Simulator you'll probably receive an error message that the URL could not be opened. Try it on your device instead.)

A word on UIApplication

You haven't used this object before, but every app has one and it handles application-wide functionality. You won't directly use `UIApplication` a lot, except for special features such as opening URLs.

> Instead, most of the time you deal with `UIApplication` is through your `AppDelegate` class, which – as you can guess from its name – is the delegate for `UIApplication`.

Finally, to load the artwork image you'll use your old friend again, the handy `UIImageView` extension.

➤ First add a new instance variable to **DetailViewController.swift**. This is necessary to be able to cancel the download task:

```
var downloadTask: URLSessionDownloadTask?
```

➤ Then add the following line to `updateUI()`:

```
if let largeURL = URL(string: searchResult.artworkLargeURL) {
  downloadTask = artworkImageView.loadImage(url: largeURL)
}
```

This is the same thing you did in `SearchResultCell`, except that you use the other artwork URL (100×100 pixels) and no placeholder image.

It's a good idea to cancel the image download if the user closes the pop-up before the image has been downloaded completely.

➤ Add a `deinit` method:

```
deinit {
  print("deinit \(self)")
  downloadTask?.cancel()
}
```

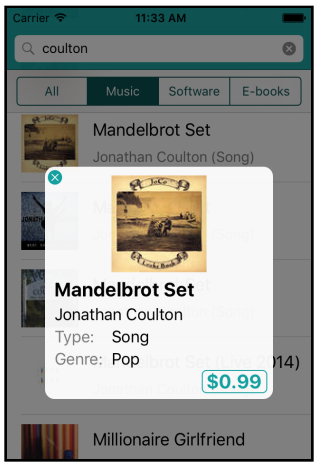
Remember that `deinit` is called whenever the object instance is deallocated and its memory is reclaimed. That happens after the user closes the `DetailViewController` and the animation to remove it from the screen has completed. If the download task is still busy by then, you cancel it.

Exercise. Why did you write `downloadTask?.cancel()` with a question mark? ■

Answer: Because `downloadTask` is an optional you need to unwrap it somehow before you can use it. When you just need to call a method on the object, it's easiest to use optional chaining like you did here.

If `downloadTask` is `nil`, there is nothing to cancel and Swift will simply ignore the call to `cancel()`.

➤ Try it out!



The pop-up now shows the artwork image

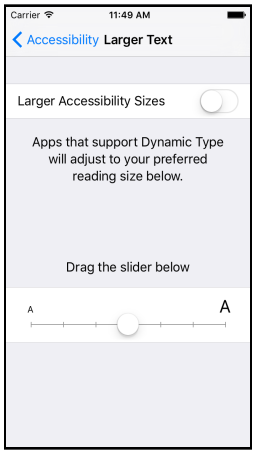
Did you see the `print()` from `deinit` after closing the pop-up? It's always a good idea to log a message when you're first trying out a new `deinit` method, to see if it really works. (If you don't see that `print()`, it means `deinit` is never called, and you may have an ownership cycle somewhere keeping your object alive longer than intended. This is why you used `[weak self]` in the closure from the `UIImageView` extension, to break any such ownership cycles.)

➤ This is a good time to commit the changes.

Dynamic Type

The iOS Settings app has an accessibility menu that allows users to choose larger or smaller text. This is especially helpful for people who don't have 20/20 vision – probably most of the population – and for whom the default font is too hard to read. Nobody likes squinting at their device!

You can find these settings under **General → Accessibility → Larger Text** on your device and also in the Simulator:

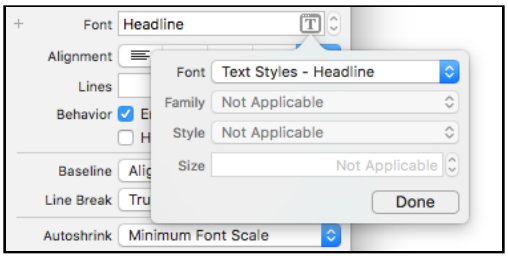


The Larger Text accessibility settings

Apps have to opt-in to use this "Dynamic Type" feature. Instead of choosing a specific font for the text labels, you use one of the built-in dynamic text styles.

Just to get some feel for how this works, you'll change the Detail pop-up to use Dynamic Type for its labels.

➤ Open the storyboard and go to the Detail View Controller scene. Change the font of the **Name** label to **Headline**:



Changing the font to the dynamic Headline style

You can't pick a size for this font. That is up to the user, based on their Larger Text settings.

➤ Choose **Editor → Size to Fit Content** to resize the label.

➤ Set the **Lines** attribute to 0. This allows the Name label to fit more than one line of text.

Of course, if you don't know beforehand how large the label's font will be, you also won't know how large the label itself will end up being, especially if it sometimes may have more than one line of text. You won't be surprised to hear that Auto Layout and Dynamic Type go hand-in-hand.

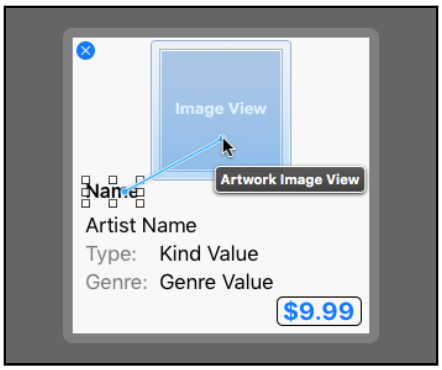
You want to make the name label resizable so that it can hold any amount of text at any possible font size, but it cannot go outside the bounds of the pop-up, nor overlap the labels below.

The trick is to capture these requirements in Auto Layout constraints.

Previously you've used the Add New Constraints button to make constraints, but that may not always give you the constraints you want. With this menu, pins are expressed as the amount of "spacing to nearest neighbor". But what exactly is the nearest neighbor?

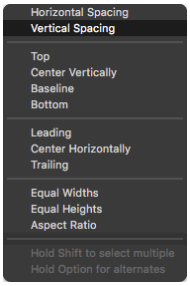
If you use the Add New Constraints button on the Name label, Interface Builder may decide to pin it to the bottom of the close button, which is weird. It makes more sense to pin the Name label to the image view instead. That's why you're going to use a different way to make constraints.

➤ Select the **Name** label. Now **Ctrl-drag** to the **Image View** and let go of the mouse button.



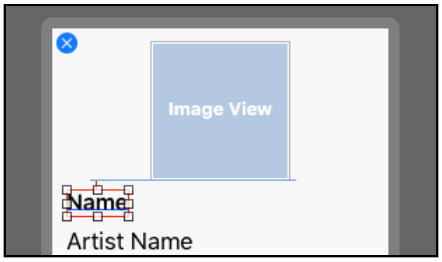
Ctrl-drag to make a new constraint between two views

From the pop-up that appears, choose **Vertical Spacing**:



The possible constraint types

This puts a vertical spacing constraint between the label and the image view:



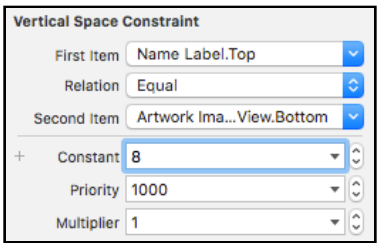
The new vertical space constraint

Of course, you'll also get some red lines because the label still needs additional constraints.

I'd like the vertical space you just added to be 8-points.

➤ Select the constraint (by carefully clicking it with the mouse or by selecting it from the outline pane), then go to the **Size inspector** and change **Constant** to 8.

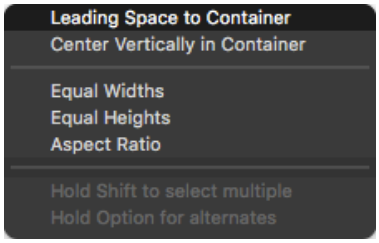
The Name label may not actually move down yet when you do this, because there are not enough constraints yet.



Attributes for the vertical space constraint

Note that the inspector clearly describes what sort of constraint this is: Name Label.Top is connected to Artwork Image View.Bottom with a distance (Constant) of 8 points.

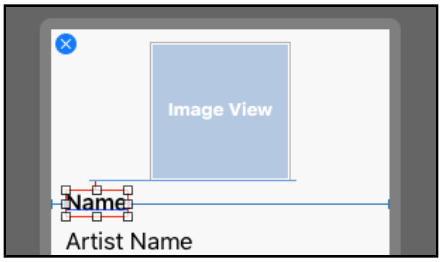
➤ Select the **Name** label again and **Ctrl-drag** to the left and connect it to **Pop-up View**. From the pop-up choose **Leading Space to Container**:



The pop-up shows different constraint types

This adds a blue bar on the left. Notice how the pop-up offered different options this time? The constraints that you can make depend on the direction that you're dragging.

➤ Repeat but this time Ctrl-drag to the right. Now choose **Trailing Space to Container**.



The constraints for the Name label

The Name label is now connected to the left edge of the Pop-up View and to its right edge – enough to determine its X-position and width – and to the bottom of the image view, for its Y-position. There is no constraint for the label's height, allowing it to grow as tall as it needs to (using the intrinsic content size).

Shouldn't these constraints be enough to uniquely determine the label's position and size? If so, why is there still a red box?

Simple: the image view now has a constraint attached to it, and therefore no longer gets automatic constraints. You also have to add constraints that give the image view its position and size.

➤ Select the **Image View**, **Ctrl-drag** up to the Pop-up View, and choose **Top Space to Container**. That takes care of the Y-position.

➤ Repeat but now choose **Center Horizontally in Container**. That center-aligns the image view to take care of the X-position. (If you don't see this option, then make sure you're not dragging outside the Pop-up View.)

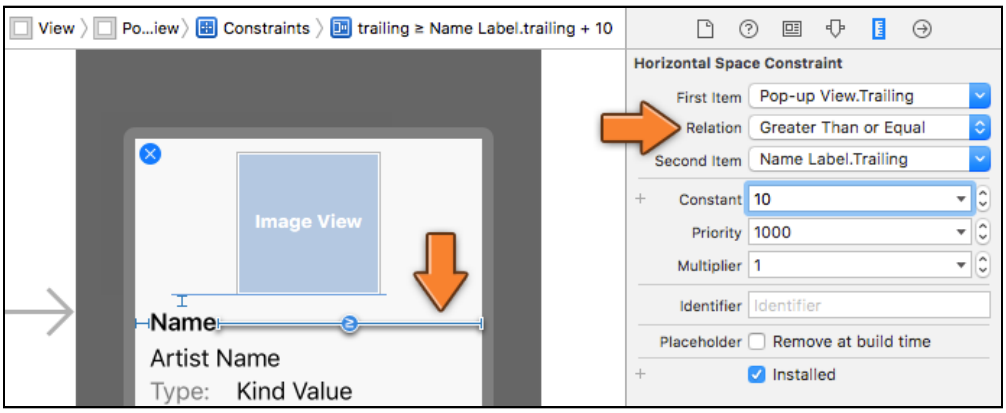
➤ Ctrl-drag again, but this time let the mouse button go while you're still inside the image view. Hold down **Shift** and put checkmarks in front of both **Width** and **Height**, then press **return**. (If you don't see both options, Ctrl-drag diagonally instead of straight up.)

Adding multiple constraints at once

Now the image view and the Name label have all blue bars. If the Name label is still misplaced for some reason – orange box – then select it and click the Update Frames button.

There's one more thing you need to fix. Look again at that blue bar on the right of the Name label. This forces the label to be always about 45 points wide. That's not what you want; instead, the label should be able to grow until it reaches the edge of the Pop-up View.

➤ Click that blue bar to select it and go to the **Size inspector**. Change **Relation** to **Greater Than or Equal**, and **Constant** to **10**.



Converting the constraint to Greater Than or Equal

Now this constraint can resize to allow the label to grow, but it can never become smaller than 10 points. This ensures there is at least a 10-point margin between the label and the edge of the Detail pop-up.

By the way, notice how this constraint is between Pop-up View.Trailing and Name Label.Trailing? In Auto Layout terminology, trailing means "on the right", while leading means "on the left".

Why didn't they just call this left and right? Well, not everyone writes in the same direction. With right-to-left languages such as Hebrew or Arabic, the meaning of trailing and leading is reversed. That allows your layouts to work without changes on these exotic languages.

➤ Run the app to try it out:

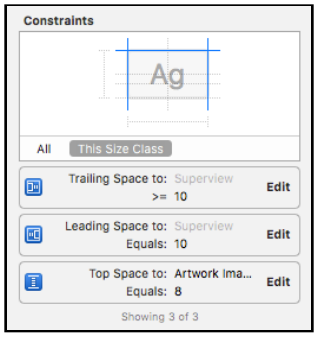
The text overlaps the other labels

Well, the word-wrapping seems to work but the text overlaps the labels below it. Let's add some more constraints so that the other labels get pushed down instead.

Tip: In the next steps I'll ask you to change the properties of the constraints using the Attributes inspector, but it can be quite tricky to select those constraints. The blue bars are often tiny, making them difficult to click. It's often easier to find the constraint in the Outline pane but it's not always immediately obvious which one you need.

A smarter way to find a constraint is to first select the view it belongs to, then go to the Size inspector and look in the Constraints section.

Here is what it looks like for the Name label:



The Name label's constraints in the Size inspector.png

To edit the constraint, double-click it or use the **Edit** button on the right.

OK, let's make those changes:

➤ Select the **Artist Name** label and change its font to **Subhead**. Give the label its ideal size with **Size to Fit Content**.

➤ Change the font of the other four labels to **Caption 1**, and **Size to Fit Content** them too. (You can do this in a single go if you multiple-select these labels by holding down the ⌘ key.)

Let's pin the **Artist Name** label. Again you do this by Ctrl-dragging.

- Pin it to the left with a Leading Space to Container.
- Pin it to the right with a Trailing Space to Container. Just like before, change this constraint's Relation to Greater Than or Equal and Constant to 10.
- Pin it to the Name label with a Vertical Spacing. Change this to size 4.

For the **Type:** label:

- Pin it to the left with a Leading Space to Container.
- On the right, pin it to the Kind Value label with a Horizontal Spacing. This should be a 20-point distance. You may get an orange label here if the original distance was larger or smaller. You'll fix that in a second.
- Pin it to the Artist Name label with a Vertical Spacing, size 8.

The **Kind Value** label is slightly different:

- Pin it to the right with a Trailing Space to Container. Change this constraint's Relation to Greater Than or Equal and Constant to 10.

- Ctrl-drag from Kind Value to Type and choose Baseline. This aligns the bottom of the text of both labels. This alignment constraint determines the Kind Value's Y-position so you don't have to make a separate constraint for that.
- With the Kind Value label selected, click the **Update Frames** button. This fixes any orange thingies.

Two more labels to go. For the **Genre:** label:

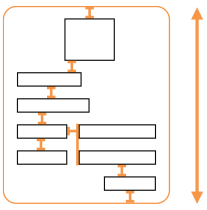
- Pin it to the left with a Leading Space to Container.
- Pin it to the Type: label with a Vertical Spacing, size 4.

And finally, the **Genre Value** label:

- Pin it to the right with a Trailing Space to Container, Greater Than or Equal 10.
- Make a Baseline alignment between Genre Value and Genre:.
- Make a Leading alignment between Genre Value and Kind Value. This keeps these two labels neatly positioned below each other.
- Resolve any Auto Layout issues. You may need to set the Constant of the alignment constraints to 0 if things don't line up properly.

That's quite a few constraints but using Ctrl-drag to make them is quite fast. With some experience you'll be able to whip together complex Auto Layout designs in no time.

There is one last thing to do. The last row of labels needs to be pinned to the price button. That way there are constraints going all the way from the top of the Pop-up View to the bottom. The heights of the labels plus the sizes of the Vertical Spacing constraints between them will now determine the height of the Detail pop-up.



The height of the pop-up view is determined by the constraints

➤ Ctrl-drag from the **$9.99** button up to **Genre Value**. Choose **Vertical Spacing**. In the Size inspector, set **Constant** to **10**.

Whoops. This messes up your carefully constructed layout and some of the constraints turn red or orange.

Exercise. Can you explain why this happens? ■

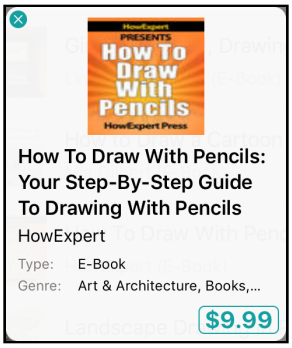
Answer: The Pop-up View still has a Height constraint that forces it to be 240 points high. But the labels and the vertical space constraints don't add up to 240.

➤ You no longer need this Height constraint, so select it (the one called **height = 240** in the outline pane) and press **delete** to get rid of it.

➤ From the **Editor → Resolve Auto Layout Issues** menu, choose **Update Frames** (from the "All Views" section).

Now all your constraints turn blue and everything fits snugly together.

➤ Run the app to try it out.

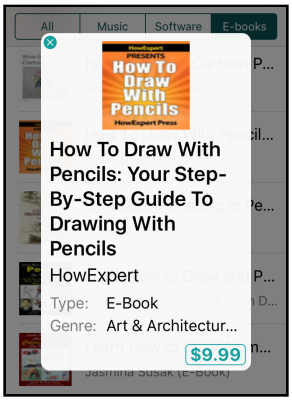


The text properly wraps without overlapping

You now have an automatically resizing Detail pop-up that uses Dynamic Type for its labels!

➤ Close the app and open the Settings app. Go to **General → Accessibility → Larger Text**. Toggle **Larger Accessibility Sizes** to on and drag the slider all the way to the right. That gives you the maximum font size (it's huge!).

Now go back to StoreSearch and open a new pop-up. The text is a lot bigger:



Changing the text size results in a bigger font

For fun, change the font of the Name label to Body. Bazinga, that's some big text!

When you're done playing, put the Name label font back to Headline, and turn off the Larger Text setting (this slider goes in the middle).

Dynamic Type is an important feature to add to your apps. This was only a short introduction but I hope the principle is clear: instead of a font with a fixed size you use one of the Text Styles: Body, Headline, Caption, and so on. Then you set up Auto Layout constraints to make your views resizable and looking good no matter how large or small the font.

➤ This is a good time to commit the changes.

Exercise. Put Dynamic Type on the cells from the table view. There's a catch: when the user returns from changing the text size settings, the app should refresh the screen without needing an app restart. You can do this by reloading the table view when the app receives a `UIContentSizeCategoryDidChange` notification (see the previous tutorial on how to handle notifications). Also check out the property `adjustsFontForContentSizeCategory` on `UILabel`. If you set this to `true`, then the app will automatically update the label whenever the font size changes. Good luck!

Check the forums at forums.raywenderlich.com for solutions from other readers. ■

Stack Views

Setting up all those constraints was quite a bit of work, but it was good Auto Layout practice! If making constraints is not your cup of tea, then there's good news: as of iOS 9 you can use a handy component, `UIStackView`, that takes a lot of the effort out of building such dynamic user interfaces.

Using stack views is fairly straightforward: you drop a **Horizontal** or **Vertical Stack View** in your scene, and then you put your labels, image views, and buttons inside that stack view. Of course, a stack view can contain other stack views as well, allowing you to create very complex layouts quite easily.

Give it a try! See if you can build the Detail pop-up with stack views. If you get stuck, we have a video tutorial series on the website that goes into great detail on `UIStackView`: raywenderlich.com/tag/stack-view

Gradients in the background

As you can see in the previous screenshots, the table view in the background is dimmed by the view of the DetailViewController, which is 50% transparent black. That allows the pop-up to stand out more.

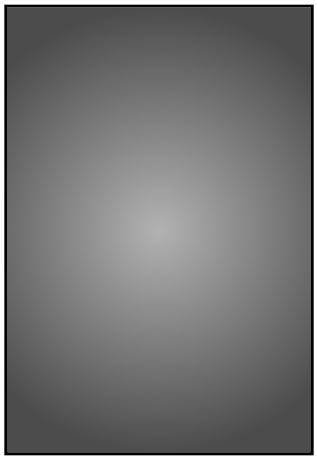
It works well, but on the other hand, a plain black overlay is a bit dull. Let's turn it into a circular gradient instead.

You could use Photoshop to draw such a gradient and place an image view behind the pop-up, but why use an image when you can also draw using Core Graphics? To pull this off, you will create your own `UIView` subclass.

➤ Add a new **Swift File** to the project. Name it **GradientView**.

This will be a very simple view. It simply draws a black circular gradient that goes from a mostly opaque in the corners to mostly transparent in the center.

Placed on a white background, it looks like this:



What the GradientView looks like by itself

➤ Replace the contents of **GradientView.swift** by:

```
import UIKit

class GradientView: UIView {
  override init(frame: CGRect) {
    super.init(frame: frame)
    backgroundColor = UIColor.clear
  }

  required init?(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)
    backgroundColor = UIColor.clear
  }

  override func draw(_ rect: CGRect) {
    // 1
    let components: [CGFloat] = [ 0, 0, 0, 0.3, 0, 0, 0, 0.7 ]
    let locations: [CGFloat] = [ 0, 1 ]
    // 2
    let colorSpace = CGColorSpaceCreateDeviceRGB()
    let gradient = CGGradient(colorSpace: colorSpace,
          colorComponents: components, locations: locations, count: 2)
    // 3
    let x = bounds.midX
    let y = bounds.midY
    let centerPoint = CGPoint(x: x, y : y)
    let radius = max(x, y)
    // 4
    let context = UIGraphicsGetCurrentContext()
    context?.drawRadialGradient(gradient!,
```

```
                                startCenter: centerPoint, startRadius: 0,
                                endCenter: centerPoint, endRadius: radius,
                                options: .drawsAfterEndLocation)
    }
  }
```

In the `init(frame)` and `init?(coder)` methods you simply set the background color to fully transparent (the "clear" color). Then in `draw()` you draw the gradient on top of that transparent background, so that it blends with whatever is below.

The drawing code uses the Core Graphics framework (also known as Quartz 2D). It may look a little scary but this is what it does:

1. First you create two arrays that contain the "color stops" for the gradient. The first color (0, 0, 0, 0.3) is a black color that is mostly transparent. It sits at location 0 in the gradient, which represents the center of the screen because you'll be drawing a circular gradient.

The second color (0, 0, 0, 0.7) is also black but much less transparent and sits at location 1, which represents the circumference of the gradient's circle. Remember that in UIKit and also in Core Graphics, colors and opacity values don't go from 0 to 255 but are fractional values between 0.0 and 1.0.

The 0 and 1 from the `locations` array represent percentages: 0% and 100%, respectively. If you have more than two colors, you can specify the percentages of where in the gradient you want to place these colors.

2. With those color stops you can create the gradient. This gives you a new `CGGradient` object.

3. Now that you have the gradient object, you have to figure out how big you need to draw it. The `midX` and `midY` properties return the center point of a rectangle. That rectangle is given by `bounds`, a `CGRect` object that describes the dimensions of the view.

If I can avoid it, I prefer not to hard-code any dimensions such as "320 by 568 points". By asking `bounds`, you can use this view anywhere you want to, no matter how big a space it should fill. You can use it without problems on any screen size from the smallest iPhone to the biggest iPad.

The `centerPoint` constant contains the coordinates for the center point of the view and `radius` contains the larger of the `x` and `y` values; `max()` is a handy function that you can use to determine which of two values is the biggest.

4. With all those preliminaries done, you can finally draw the thing. Core Graphics drawing always takes places in a so-called graphics context. We're not going to worry about exactly what that is, just know that you need to obtain a reference to the current context and then you can do your drawing.

And finally, the `drawRadialGradient()` function draws the gradient according to your specifications.

It generally speaking isn't optimal to create new objects inside your `draw()` method, such as gradients, especially if `draw()` is called often. In that case it is better to create the objects the first time you need them and to reuse the same instance over and over (lazy loading!).

However, you don't really have to do that here because this `draw()` method will be called just once – when the `DetailViewController` gets loaded – so you can get away with being less than optimal.

> **Note:** By the way, you'll only be using `init(frame)` to create the `GradientView` instance. The other init method, `init?(coder)`, is never used in this app. `UIView` demands that all subclasses implement `init?(coder)` – that is why it is marked as `required` – and if you remove this method, Xcode will give an error.

Putting this new `GradientView` class into action is pretty easy. You'll add it to your own presentation controller object. That way the `DetailViewController` doesn't need to know anything about it. Dimming the background is really a side effect of doing a presentation, so it belongs in the presentation controller.

➤ Open **DimmingPresentationController.swift** and add the following code inside the class:

```
lazy var dimmingView = GradientView(frame: CGRect.zero)

override func presentationTransitionWillBegin() {
  dimmingView.frame = containerView!.bounds
  containerView!.insertSubview(dimmingView, at: 0)
}
```

The `presentationTransitionWillBegin()` method is invoked when the new view controller is about to be shown on the screen. Here you create the `GradientView` object, make it as big as the `containerView`, and insert it behind everything else in this "container view".

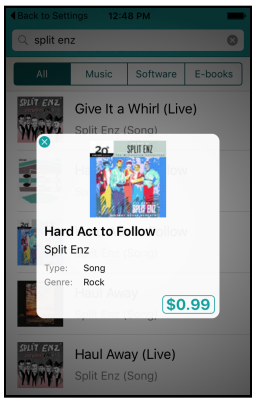
The container view is a new view that is placed on top of the `SearchViewController`, and it contains the views from the `DetailViewController`. So this piece of logic places the `GradientView` in between those two screens.

There's one more thing to do: because the `DetailViewController`'s background color is still 50% black, this color gets multiplied with the colors inside the gradient view, making the gradient look extra dark. It's better to set the background color to 100% transparent, but if we do that in the storyboard it makes it harder to see and edit the pop-up view. So let's do this in code instead.

➤ Add the following line to **DetailViewController.swift**'s `viewDidLoad()`:

```
view.backgroundColor = UIColor.clear
```

➤ Run the app and see what happens.



The background behind the pop-up now has a gradient

Nice, that looks a lot smarter.

Animation!

The pop-up itself looks good already, but the way it enters the screen – Poof! It's suddenly there – is a bit unsettling. iOS is supposed to be the king of animation, so let's make good on that.

You've used Core Animation and UIView animations before. This time you'll use a so-called **keyframe animation** to make the pop-up bounce into view.

To animate the transition between two screens, you use an animation controller object. The purpose of this object is to animate a screen while it's being presented or dismissed, nothing more.

Now let's add some liveliness to this pop-up!

➤ Add a new Swift File to the project, named **BounceAnimationController**.

➤ Replace the contents of this new file with:

```
import UIKit

class BounceAnimationController: NSObject,
                                 UIViewControllerAnimatedTransitioning {

  func transitionDuration(using transitionContext:
                  UIViewControllerContextTransitioning?) -> TimeInterval {
    return 0.4
  }
```

```
  func animateTransition(using transitionContext:
                                 UIViewControllerContextTransitioning) {

    if let toViewController = transitionContext.viewController(
                        forKey: UITransitionContextViewControllerKey.to),
       let toView = transitionContext.view(
                                forKey: UITransitionContextViewKey.to) {

      let containerView = transitionContext.containerView
      toView.frame = transitionContext.finalFrame(for: toViewController)
      containerView.addSubview(toView)
      toView.transform = CGAffineTransform(scaleX: 0.7, y: 0.7)

      UIView.animateKeyframes(
            withDuration: transitionDuration(using: transitionContext),
            delay: 0, options: .calculationModeCubic, animations: {
        UIView.addKeyframe(withRelativeStartTime: 0.0,
                           relativeDuration: 0.334, animations: {
          toView.transform = CGAffineTransform(scaleX: 1.2, y: 1.2)
        })
        UIView.addKeyframe(withRelativeStartTime: 0.334,
                           relativeDuration: 0.333, animations: {
          toView.transform = CGAffineTransform(scaleX: 0.9, y: 0.9)
        })
        UIView.addKeyframe(withRelativeStartTime: 0.666,
                           relativeDuration: 0.333, animations: {
          toView.transform = CGAffineTransform(scaleX: 1.0, y: 1.0)
        })
      }, completion: { finished in
        transitionContext.completeTransition(finished)
      })
    }
  }
}
```

To become an animation controller, the object needs to extend `NSObject` and also implement the `UIViewControllerAnimatedTransitioning` protocol – quite a mouthful! The important methods from this protocol are:

- `transitionDuration(…)` – This determines how long the animation is. You're making the pop-in animation last for only 0.4 seconds but that's long enough. Animations are fun but they shouldn't keep the user waiting.
- `animateTransition(…)` – This performs the actual animation.

To find out what to animate, you look at the `transitionContext` parameter. This gives you a reference to new view controller and lets you know how big it should be.

The actual animation starts at the line `UIView.animateKeyframes(…)`. This works like all UIView-based animation: you set the initial state before the animation block, and UIKit will automatically animate any properties that get changed inside the closure. The difference with before is that a keyframe animation lets you animate the view in several distinct stages.

The property you're animating is the `transform`. If you've ever taken any matrix math you'll be pleased – or terrified! – to hear that this is an affine transformation matrix. It allows you to do all sorts of funky stuff with the view, such as rotating or shearing it, but the most common use of the transform is for scaling.

The animation consists of several **keyframes**. It will smoothly proceed from one keyframe to the next over a certain amount of time. Because you're animating the view's scale, the different `toView.transform` values represent how much bigger or smaller the view will be over time.

The animation starts with the view scaled down to 70% (scale 0.7). The next keyframe inflates it to 120% its normal size. After that, it will scale the view down a bit again but not as much as before (only 90% of its original size). The final keyframe ends up with a scale of 1.0, which restores the view to an undistorted shape. By quickly changing the view size from small to big to small to normal, you create a bounce effect.

You also specify the duration between the successive keyframes. In this case, each transition from one keyframe to the next takes 1/3rd of the total animation time. These times are not in seconds but in fractions of the animation's total duration (0.4 seconds). Feel free to mess around with the animation code. No doubt you can make it much more spectacular!

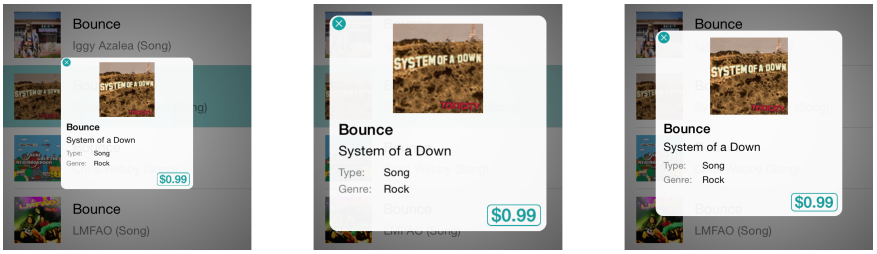
To make this animation happen you have to tell the app to use the new animation controller when presenting the Detail pop-up. That happens in the transitioning delegate inside **DetailViewController.swift**.

➤ Inside the `UIViewControllerTransitioningDelegate` extension, add the following method:

```
func animationController(forPresented presented: UIViewController,
                presenting: UIViewController, source: UIViewController)
                -> UIViewControllerAnimatedTransitioning? {
  return BounceAnimationController()
}
```

And that's all you need to do.

➤ Run the app and get ready for some bouncing action!



The pop-up animates

The pop-up looks a lot spiffier with the bounce animation but there are two things that could be better: the `GradientView` still appears abruptly in the background, and the animation upon dismissal of the pop-up is very plain.

There's no reason why you cannot have two things animating at the same time, so let's make the `GradientView` fade in while the pop-up bounces into view. That is a job for the presentation controller, because that's what provides the gradient view.

➤ Go to **DimmingPresentationController.swift** and add the following to the bottom of `presentationTransitionWillBegin()`:

```
dimmingView.alpha = 0
if let coordinator = presentedViewController.transitionCoordinator {
  coordinator.animate(alongsideTransition: { _ in
  self.dimmingView.alpha = 1
  }, completion: nil)
}
```

You set the alpha value of the gradient view to 0 to make it completely transparent and then animate it back to 1 (or 100%) and fully visible, resulting in a simple fade-in. That's a bit more subtle than making the gradient appear so abruptly.

The special thing here is the `transitionCoordinator` stuff. This is the UIKit traffic cop in charge of coordinating the presentation controller and animation controllers and everything else that happens when a new view controller is presented.

The important thing to know about the `transitionCoordinator` is that any of your animations should be done in a closure passed to `animateAlongsideTransition` to keep the transition smooth. If your users wanted choppy animations, they would have bought Android phones!

➤ Also add the method `dismissalTransitionWillBegin()`, which is used to animate the gradient view out of sight when the Detail pop-up is dismissed:

```
override func dismissalTransitionWillBegin()  {
  if let coordinator = presentedViewController.transitionCoordinator {
    coordinator.animate(alongsideTransition: { _ in
      self.dimmingView.alpha = 0
    }, completion: nil)
  }
}
```

This does the inverse: it animates the alpha value back to 0% to make the gradient view fade out.

➤ Run the app. The dimming gradient now appears almost without you even noticing it. Slick!

Let's add one more quick animation because this stuff is just too much fun. :-)

After tapping the Close button the pop-up slides off the screen, like modal screens always do. Let's make this a bit more exciting and make it slide up instead of down.

For that you need another animation controller.

➤ Add a new Swift File to the project, named **SlideOutAnimationController**.

➤ Replace the contents with:

```
import UIKit

class SlideOutAnimationController: NSObject,
                                   UIViewControllerAnimatedTransitioning {
  func transitionDuration(using transitionContext:
                UIViewControllerContextTransitioning?) -> TimeInterval {
    return 0.3
  }

  func animateTransition(using transitionContext:
                                   UIViewControllerContextTransitioning) {
    if let fromView = transitionContext.view(forKey:
                                       UITransitionContextViewKey.from) {
      let containerView = transitionContext.containerView
      let duration = transitionDuration(using: transitionContext)
      UIView.animate(withDuration: duration, animations: {
        fromView.center.y -= containerView.bounds.size.height
        fromView.transform = CGAffineTransform(scaleX: 0.5, y: 0.5)
      }, completion: { finished in
        transitionContext.completeTransition(finished)
      })
    }
  }
}
```

This is pretty much the same as the other animation controller, except that the animation itself is different. Inside the animation block you subtract the height of the screen from the view's center position while simultaneously zooming it out to 50% of its original size, making the Detail screen fly up-up-and-away.

➤ In **DetailViewController.swift**, add the following method to the `UIViewControllerTransitioningDelegate` extension:

```
func animationController(forDismissed dismissed: UIViewController)
                            -> UIViewControllerAnimatedTransitioning? {
  return SlideOutAnimationController()
}
```

➤ Run the app and try it out. That looks pretty sweet if you ask me!

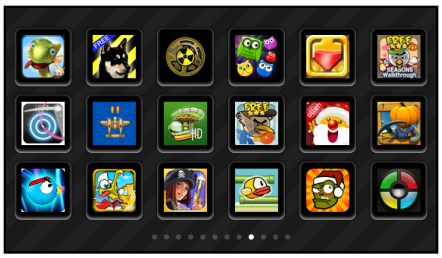
➤ If you're happy with the way the animation looks, then commit your changes.

You can find the project files for the app up to this point under **06 - Detail Pop-up** in the tutorial's Source Code folder.

Exercise. Create some exciting new animations. I'm sure you can improve on mine. Hint: use the `transform` matrix to add some rotation into the mix. ■

Fun with landscape

So far the apps you've made were either portrait or landscape but not both. Let's change the app so that it shows a completely different user interface when you flip the device over. When you're done, the app will look like this:



The app looks completely different in landscape orientation

The landscape screen shows just the artwork for the search results. Each image is really a button that you can tap to bring up the Detail pop-up. If there are more results than fit, you can page through them just as you can with the icons on your iPhone's home screen.

The to-do list for this section is:

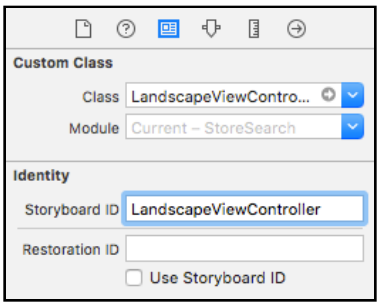
- Create a new view controller and show that when the device is rotated. Hide this view controller when the device returns to the portrait orientation.
- Put some fake buttons in a `UIScrollView`, in order to learn how to use scroll views.
- Add the paging control (the dots at the bottom) so you can page through the contents of the scroll view.
- Put the artwork images on the buttons. You will have to download these images from the iTunes server.
- When the user taps a button, show the Detail pop-up.

Let's begin by creating a very simple view controller that shows just a text label.

➤ Add a new file to the project using the **Cocoa Touch Class** template. Name it **LandscapeViewController** and make it a subclass of **UIViewController**.

➤ In Interface Builder, drag a new **View Controller** into the canvas; put it below the Search View Controller.

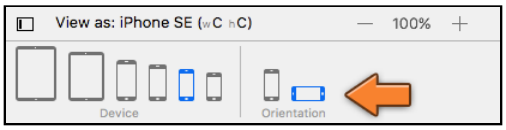
➤ In the Identity inspector, change the **Class** to **LandscapeViewController**. Also type this into the **Storyboard ID** field.



Giving the view controller an ID

There will be no segue to this view controller. You'll instantiate this view controller programmatically when you detect a device rotation, and for that it needs to have an ID so you can look it up in the storyboard.

➤ Use the **View as:** panel to change the orientation to landcape.



Changing Interface Builder to landscape

This flips *all* the scenes in the storyboard to landscape but that is OK – it doesn't change what happens when you run the app. Putting Interface Builder in landscape mode just provides a design aid that makes it easier to layout your UI. What actually happens when you run the app depends on the orientation the user is holding the device.

The trick is to use Auto Layout constraints to make sure that the view controllers properly resize to landscape or portrait at runtime.

➤ Change the **Background** of the view to **Black Color**.

➤ Drag a new **Label** into the scene and give it some text. You're just using this label to verify that the new view controller shows up in the correct orientation.

➤ Use the **Align** button to make horizontal and vertical centering constraints for the label.

Your design should look something like this:

Initial design for the landscape view controller

As you know by now, view controllers have a bunch of methods that are invoked by UIKit at given times, such as `viewDidLoad()`, `viewWillAppear()`, and so on. There is also a method that is invoked when the device is flipped over. You can override this method to show (and hide) the new `LandscapeViewController`.

➤ Add the following method to **SearchViewController.swift**:

```
override func willTransition(to newCollection: UITraitCollection,
             with coordinator: UIViewControllerTransitionCoordinator) {
  super.willTransition(to: newCollection, with: coordinator)

  switch newCollection.verticalSizeClass {
  case .compact:
    showLandscape(with: coordinator)
  case .regular, .unspecified:
    hideLandscape(with: coordinator)
  }
}
```

This method isn't just invoked on device rotations but any time the **trait collection** for the view controller changes. So what is a trait collection? It is, um, a collection of **traits**, where a trait can be:

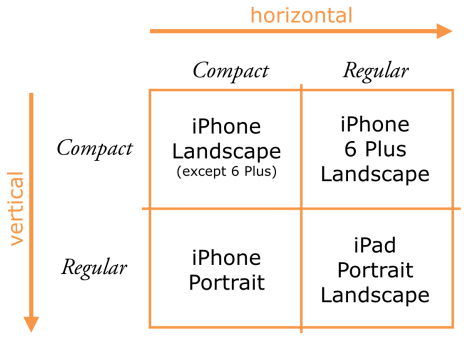
- the horizontal size class
- the vertical size class
- the display scale (is this a Retina screen or not?)
- the user interface idiom (is this an iPhone or iPad?)
- the preferred Dynamic Type font size
- and a few other things

Whenever one or more of these traits change, for whatever reason, UIKit calls `willTransition(to:with:)` to give the view controller a chance to adapt to the new traits.

What we are interested in here are the **size classes**. This feature allows you to design a user interface that is independent of the device's actual dimensions or orientation. With size classes, you can create a single storyboard that works across all devices, from iPhone to iPad – a so-called "universal storyboard".

So how exactly do these size classes work? Well, there's two of them, a horizontal one and a vertical one, and each can have two values: *compact* or *regular*.

The combination of these four things creates the following possibilities:



Horizontal and vertical size classes

When an iPhone app is in portrait orientation, the horizontal size class is *compact* and the vertical size class is *regular*.

Upon a rotation to landscape, the vertical size class changes to *compact*.

What you may not have expected is that the horizontal size class doesn't change and stays *compact* in both portrait and landscape orientations – except on the iPhone 6s Plus and 7 Plus, that is.

In landscape, the horizontal size class on the Plus is *regular*. That's because the larger dimensions of the iPhone 6s Plus and 7 Plus can fit a split screen in landscape mode, like the iPad (something you'll see later on).

What this boils down to is that to detect an iPhone rotation, you just have to look at how the vertical size class changed. That's exactly what the `switch` statement does:

```
switch newCollection.verticalSizeClass {
case .compact:
  showLandscape(with: coordinator)
case .regular, .unspecified:
  hideLandscape(with: coordinator)
}
```

If the new vertical size class is `.compact` the device got flipped to landscape and you show the `LandscapeViewController`. But if the new size class is `.regular`, the app is back in portrait and you hide the landscape view again.

The reason the second `case` statement also checks `.unspecified` is because `switch` statements must always be exhaustive and have cases for all possible values. `.unspecified` shouldn't happen but just in case it does, you also hide the landscape view. This is another example of defensive programming.

Just to keep things readable, the actual showing and hiding happens in methods of their own. You will add these next.

In the early years of iOS it was tricky to put more than one view controller on the same screen. The motto used to be: one screen, one view controller. However, on devices with larger screens such as the iPad that became inconvenient.

You often want one area of the screen to be controlled by one view controller and a second area by its own view controller – so now view controllers are allowed to be part of other view controllers if you follow a few rules.

This is called **view controller containment**. These APIs are not limited to just the iPad; you can take advantage of them on the iPhone as well. These days a view controller is no longer expected to manage a screenful of content, but manages a "self-contained presentation unit", whatever that is for your app.

You're going to use view controller containment for the `LandscapeViewController`.

It would be perfectly possible to make a modal segue to this scene and use your own presentation and animation controllers for the transition. But you've already done that and it's more fun to play with something new. Besides, it's useful to learn about containment and child view controllers.

➤ Add an instance variable to **SearchViewController.swift**:

```
var landscapeViewController: LandscapeViewController?
```

This is an optional because there will only be an active `LandscapeViewController` instance if the phone is in landscape orientation. In portrait this will be `nil`.

➤ Add the following method:

```
func showLandscape(with coordinator:
                                UIViewControllerTransitionCoordinator) {
  // 1
  guard landscapeViewController == nil else { return }
  // 2
  landscapeViewController = storyboard!.instantiateViewController(
                              withIdentifier: "LandscapeViewController")
                              as? LandscapeViewController
  if let controller = landscapeViewController {
    // 3
    controller.view.frame = view.bounds
    // 4
    view.addSubview(controller.view)
    addChildViewController(controller)
    controller.didMove(toParentViewController: self)
```

```
    }
  }
```

In previous tutorials you would call `present(animated, completion)` or make a segue to show the new modal screen. Here, however, you add the new `LandscapeViewController` as a *child* view controller of `SearchViewController`.

Here's how it works, step-by-step:

1. It should never happen that the app instantiates a second landscape view when you're already looking at one. The guard that `landscapeViewController` is still `nil` codifies this requirement. If it should happen that this condition doesn't hold – we're already showing the landscape view – then we simply `return` right away.

2. Find the scene with the ID "LandscapeViewController" in the storyboard and instantiate it. Because you don't have a segue you need to do this manually. This is why you filled in that Storyboard ID field in the Identity inspector.

The `landscapeViewController` instance variable is an optional so you need to unwrap it before you can continue.

3. Set the size and position of the new view controller. This makes the landscape view just as big as the `SearchViewController`, covering the entire screen.

The `frame` is the rectangle that describes the view's position and size in terms of its superview. To move a view to its final position and size you usually set its `frame`. The `bounds` is also a rectangle but seen from the inside of the view.

Because `SearchViewController`'s view is the superview here, the `frame` of the landscape view must be made equal to the `SearchViewController`'s `bounds`.

4. These are the minimum required steps to add the contents of one view controller to another, in this order:

a. First, add the landscape controller's view as a subview. This places it on top of the table view, search bar and segmented control.

b. Then tell the `SearchViewController` that the `LandscapeViewController` is now managing that part of the screen, using `addChildViewController()`. If you forget this step then the new view controller may not always work correctly.

c. Tell the new view controller that it now has a parent view controller with `didMove(toParentViewController)`.

In this new arrangement, `SearchViewController` is the "parent" view controller, and `LandscapeViewController` is the "child". In other words, the Landscape screen is embedded inside the `SearchViewController`.

Note: Even though it will appear on top of everything else, the Landscape screen is not presented modally. It is "contained" in its parent view controller, and therefore owned and managed by it, not independent like a modal screen. This is an important distinction.

View controller containment is also used for navigation and tab bar controllers where the `UINavigationController` and `UITabBarController` "wrap around" their child view controllers.

Usually when you want to show a view controller that takes over the whole screen you'd use a modal segue. But when you want just a portion of the screen to be managed by its own view controller you'd make it a child view controller.

One of the reasons you're not using a modal segue for the Landscape screen in this app, even though it is a full-screen view controller, is that the Detail pop-up already is modally presented and this could potentially cause conflicts. Besides, I wanted to show you a fun alternative to modal segues.

➤ To get the app to compile, add an empty implementation of the "hide" method:

```
func hideLandscape(with coordinator:
                                UIViewControllerTransitionCoordinator) {
}
```

By the way, the transition coordinator parameter is needed for doing animations, which you'll add soon.

➤ Try it out! Run the app, do a search and flip over your iPhone or the Simulator to landscape.

The Simulator after flipping to landscape

Remember: to rotate the Simulator, press ⌘ and the arrow keys. It's possible that the Simulator won't flip over right away – it can be buggy like that. When that happens, press **⌘+arrow key** a few more times.

This is not doing any animation yet. As always, first get it to work and only then make it look pretty.

If you don't do a search first before rotating to landscape, the keyboard may remain visible. You'll fix that shortly. In the mean time you can press ⌘**+K** to hide the keyboard manually.

Flipping back to portrait doesn't work yet but that's easily fixed.

➤ Implement the method that will hide the landscape view controller:

```
func hideLandscape(with coordinator:
                                UIViewControllerTransitionCoordinator) {
  if let controller = landscapeViewController {
    controller.willMove(toParentViewController: nil)
    controller.view.removeFromSuperview()
    controller.removeFromParentViewController()
    landscapeViewController = nil
  }
}
```

This is essentially the inverse of what you did to embed the view controller.

First you call `willMove(toParentViewController: nil)` to tell the view controller that it is leaving the view controller hierarchy (it no longer has a parent), then you remove its view from the screen, and finally `removeFromParentViewController()` truly disposes of the view controller.

You also set the instance variable to `nil` in order to remove the last strong reference to the `LandscapeViewController` object now that you're done with it.

➤ Run the app. Flipping back to portrait should remove the black landscape view again.

Note: If you press ⌘**-right** twice, the Simulator first rotates to landscape and then to portrait, but the `LandscapeViewController` does not disappear. Why is that?

It's a bit hard to see in the Simulator, but what you're looking at now is *not* portrait but portrait upside down. This orientation is not recognized by the app (see the Device Orientation setting under Deployment Info in the project settings) and therefore it keeps thinking it's in landscape.

Press ⌘**-right** twice again and you're back in regular portrait.

Whenever I write a new view controller, I like to put a `print()` in its `deinit` method just to make sure the object is properly deallocated when the screen closes.

➤ Add a `deinit` method to **LandscapeViewController.swift**:

```
deinit {
  print("deinit \(self)")
}
```

➤ Run the app and verify that `deinit` is indeed being called after rotating back to portrait.

The transition to the landscape view is a bit abrupt. I don't want to go overboard with animations here as the screen is already doing a rotating animation. A simple crossfade will be sufficient.

➤ Change the `showLandscape(with)` method to:

```
func showLandscape(with coordinator:
                               UIViewControllerTransitionCoordinator) {
  . . .
  if let controller = landscapeViewController {
    controller.view.frame = view.bounds
    controller.view.alpha = 0

    view.addSubview(controller.view)
    addChildViewController(controller)

    coordinator.animate(alongsideTransition: { _ in
      controller.view.alpha = 1
    }, completion: { _ in
      controller.didMove(toParentViewController: self)
    })
  }
}
```

You're still doing the same things as before, except now the landscape view starts out completely see-through (`alpha` = 0) and slowly fades in while the rotation takes place until it's fully visible (`alpha` = 1).

Now you see why the `UIViewControllerTransitionCoordinator` object is needed, so your animation can be performed alongside the rest of the transition from the old traits to the new. This ensures the animations run as smoothly as possible.

The call to `animate(alongsideTransition, completion)` takes two closures: the first is for the animation itself, the second is a "completion handler" that gets called after the animation finishes. The completion handler gives you a chance to delay the call to `didMove(toParentViewController)` until the animation is over.

Both closures are given a "transition coordinator context" parameter (the same context that animation controllers get) but it's not very interesting here and you use the _ wildcard to ignore it.

> **Note:** You don't have to write `self.controller` inside these closures because `controller` is not an instance variable. It is a local constant that is valid only inside the `if let` statement. `self` is only used to refer to instance variables and methods, or the view controller object itself, but is never used for locals.

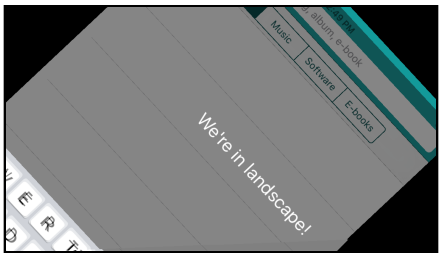
➤ Make likewise changes to `hideLandscape(with)`:

```
func hideLandscape(with coordinator:
                                UIViewControllerTransitionCoordinator) {
  if let controller = landscapeViewController {
    controller.willMove(toParentViewController: nil)

    coordinator.animate(alongsideTransition: { _ in
      controller.view.alpha = 0
    }, completion: { _ in
      controller.view.removeFromSuperview()
      controller.removeFromParentViewController()
      self.landscapeViewController = nil
    })
  }
}
```

This time you fade out the view (back to `alpha` = 0). You don't remove the view and the controller until the animation is completely done.

➤ Try it out. The transition between the portrait and landscape views should be a lot smoother now.



The transition from portrait to landscape

Tip: To see the transition animation in slow motion, select **Debug → Slow Animations** from the Simulator menu bar.

Note: The order of operations for removing a child view controller is exactly the other way around from adding a child view controller, except for the calls to `willMove` and `didMove(toParentViewController)`.

The rules for view controller containment say that when adding a child view controller, the last step is to call `didMove(toParentViewController)`. UIKit does not know when to call this method, as that needs to happen after any of your animations. You are responsible for sending the "did move to parent" message to the child view controller once the animation completes.

There is also a `willMove(toParentViewController)` but that gets called on your behalf by `addChildViewController()` already, so you're not supposed to do that yourself.

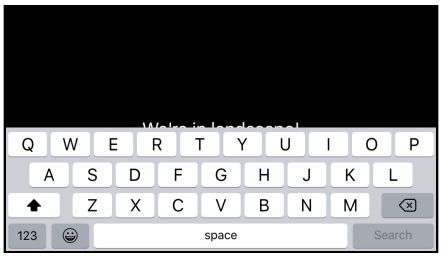
The rules are opposite when removing the child controller. First you should call `willMove(toParentViewController: nil)` to let the child view controller know

that it's about to be removed from its parent. The child view controller shouldn't actually be removed until the animation completes, at which point you call `removeFromParentViewController()`. That method will then take care of sending the "did move to parent" message.

You can find these rules in the API documentation for `UIViewController`.

Hiding the keyboard and pop-up

There are two more small tweaks to make. Maybe you already noticed that when rotating the app while the keyboard is showing, the keyboard doesn't go away.



The keyboard is still showing in landscape mode

Exercise. See if you can fix that yourself. ■

Answer: You've done something similar already after the user taps the Search button. The code is exactly the same here.

➤ Add the following line to `showLandscape(with)`:

```
func showLandscape(with coordinator:
                              UIViewControllerTransitionCoordinator) {
    . . .
    coordinator.animate(alongsideTransition: { _ in
      controller.view.alpha = 1
      self.searchBar.resignFirstResponder()          // add this line
    }, completion: { _ in
      . . .
    })
  }
}
```

Now the keyboard disappears as soon as you flip the device. I found it looks best if you call `resignFirstResponder()` inside the animate-alongside-transition closure. After all, hiding the keyboard also happens with an animation.

Speaking of things that stay visible, what happens when you tap a row in the table view and then rotate to landscape? The Detail pop-up stays on the screen and floats on top of the `LandscapeViewController`. I find that a little strange. It would be better if the app dismissed the pop-up before rotating.

Exercise. See if you can fix that one. ■

Answer: The Detail pop-up is presented modally with a segue, so you can call `dismiss(animated, completion)` to dismiss it, just like you do in the `close()` action method.

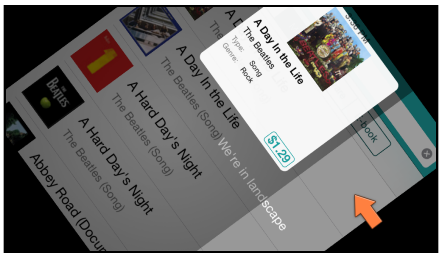
There's a complication: you should only dismiss the Detail screen when it is actually visible. For that you can look at the `presentedViewController` property. This returns a reference to the current modal view controller, if any. If `presentedViewController` is `nil` there isn't anything to dismiss.

➤ Add the following code inside the `animate(alongsideTransition)` closure in `showLandscape(with)`:

```
if self.presentedViewController != nil {
  self.dismiss(animated: true, completion: nil)
}
```

➤ Run the app and tap on a search result, then flip to landscape. The pop-up should now fly off the screen. When you return to portrait, the pop-up is nowhere to be seen.

If you look really carefully while the screen rotates, you can see a glitch at the right side of the screen. The gradient view doesn't appear to stretch to fill up the extra space:



There is a gap next to the gradient view

(Press ⌘**+T** to turn on slow animations in the Simulator so you can clearly see this happening.)

It's only a small detail but we can't have such imperfections in our apps!

The solution is to pin the `GradientView` to the edges of the window so that it will always stretch along with it. But you didn't create `GradientView` in Interface Builder… so how do you give it constraints?

It is possible to create constraints in code, using the `NSLayoutConstraint` class, but there is an easier solution: you can simply change the `GradientView`'s autoresizing behavior.

Autoresizing is what iOS developers used before Auto Layout existed. It's simpler to use but also less powerful. You've already used autoresizing in the MyLocations app where you enabled or disabled the different "springs and struts" for your views in Interface Builder. It's very easy to do the same thing from code.

Using the `autoresizingMask` property you can tell a view what it should do when its superview changes size. You have a variety of options, such as: do nothing, stick to a certain edge of the superview, or change in size proportionally.

The possibilities are much more limited than what you can do with Auto Layout, but for many scenarios autoresizing is good enough.

The easiest place to set this autoresizing mask is in `GradientView`'s init methods.

➤ Add the following line to `init(frame)` and `init?(coder)` in **GradientView.swift**:

```
autoresizingMask = [.flexibleWidth , .flexibleHeight]
```

This tells the view that it should change both its width and its height proportionally when the superview it belongs to resizes (due to being rotated or otherwise).

In practice this means the `GradientView` will always cover the same area that its superview covers and there should be no more gaps, even if the device gets rotated.

➤ Try it out! The gradient now always covers the whole screen.

The Detail pop-up flying up and out the screen looks a little weird in combination with the rotation animation. There's too much happening on the screen at once, to my taste. Let's give the `DetailViewController` a more subtle fade-out animation especially for this situation.

When you tap the X button to dismiss the pop-up, you'll still make it fly out of the screen. But when it is automatically dismissed upon rotation, the pop-up will fade out with the rest of the table view instead.

You'll give `DetailViewController` a property that that specifies how it will animate the pop-up's dismissal. You can use an `enum` for this.

➤ Add the following to **DetailViewController.swift**, *inside* the class:

```
enum AnimationStyle {
  case slide
  case fade
}

var dismissAnimationStyle = AnimationStyle.fade
```

This defines a new `enum` named `AnimationStyle`. An enum, or enumeration, is simply a list of possible values. The `AnimationStyle` enum has two values, `slide` and `fade`. Those are the animations the Detail pop-up can perform when dismissed.

The `dismissAnimationStyle` variable determines which animation is chosen. This variable is of type `AnimationStyle`, so it can only contain one of the values from that enum. By default it is `.fade`, the animation that will be used when rotating to landscape.

> **Note:** The full name of the enum is `DetailViewController.AnimationStyle` because it sits inside the `DetailViewController` class.
>
> It's a good idea to keep the things that are closely related to a particular class, such as this enum, inside the definition for that class. That puts them inside the class's *namespace*.
>
> Doing this allows you to also add a completely different `AnimationStyle` enum to one of the other view controllers, without running into naming conflicts.

➤ In the `close()` method set the animation style to `.slide`, so that this keeps using the animation you're already familiar with:

```
@IBAction func close() {
  dismissAnimationStyle = .slide
  dismiss(animated: true, completion: nil)
}
```

➤ In the extension for the transitioning delegate, change the method that vends the animation controller for dismissing the pop-up to the following:

```
func animationController(forDismissed dismissed: UIViewController) ->
                                  UIViewControllerAnimatedTransitioning? {
  switch dismissAnimationStyle {
  case .slide:
    return SlideOutAnimationController()
  case .fade:
    return FadeOutAnimationController()
  }
}
```

Instead of always returning a new `SlideOutAnimationController` instance, it now looks at the value from `dismissAnimationStyle`. If it is `.fade`, then it returns a new `FadeOutAnimationController` object. You still have to write that class.

➤ Add a new **Swift File** to the project, named **FadeOutAnimationController**.

➤ Replace the source code of that new file with:

```
import UIKit

class FadeOutAnimationController: NSObject,
                                  UIViewControllerAnimatedTransitioning {
  func transitionDuration(using transitionContext:
                UIViewControllerContextTransitioning?) -> TimeInterval {
    return 0.4
```

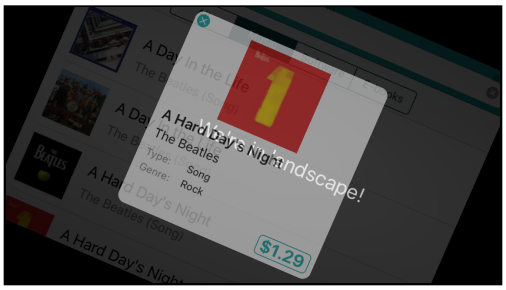
```
    }

    func animateTransition(using transitionContext:
                                  UIViewControllerContextTransitioning) {
      if let fromView = transitionContext.view(
                                forKey: UITransitionContextViewKey.from) {
        let duration = transitionDuration(using: transitionContext)
        UIView.animate(withDuration: duration, animations: {
          fromView.alpha = 0
        }, completion: { finished in
          transitionContext.completeTransition(finished)
        })
      }
    }
  }
```

This is mostly the same as the other animation controllers. The actual animation simply sets the view's `alpha` value to 0 in order to fade it out.

➤ Run the app, bring up the Detail pop-up and rotate to landscape.

The pop-up should now fade out while the landscape view fades in. (Enable slow animations to clearly see what is going on.)



The pop-up fades out instead of flying away

And that does it. If you wanted to create more animations that can be used on dismissal, you only have to add a new value to the `AnimationStyle` enum and check for it in the `animationController(forDismissed)` method. And build a new animation controller, of course.

That concludes the first version of the landscape screen. It doesn't do much yet, but it's already well integrated with the rest of the app. That's worthy of a commit, methinks.

Adding the scroll view

If an app has more to show than can fit on the screen, you can use a **scroll view**, which allows the user to drag the content up and down or left and right.

You've already been working with scroll views all this time without knowing it: the `UITableView` object extends from `UIScrollView`.

In this section you're going to use a scroll view of your own, in combination with a **paging control**, so you can show the artwork for all the search results even if there are more images than fit on the screen at once.

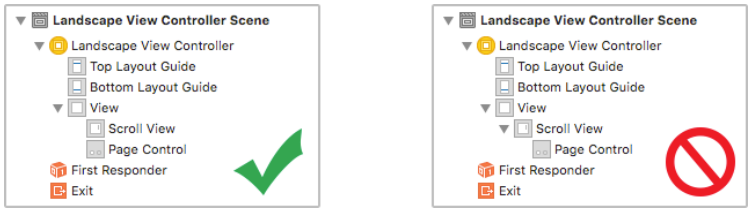
➤ Open the storyboard and delete the label from the Landscape View Controller.

➤ Now drag a new **Scroll View** into the scene. Make it as big as the screen (568 by 320 points in landscape).

➤ Drag a new **Page Control** object into the scene (make sure you pick Page Control and not Page View Controller).

This gives you a small view with three white dots. Place it bottom center. The exact location doesn't matter because you'll move it to the right position later.

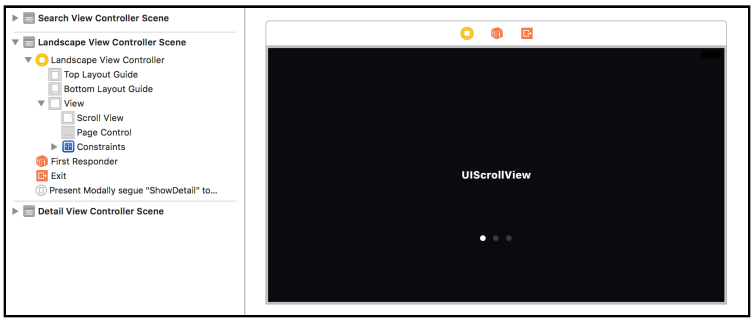
Important: Do not place the Page Control *inside* the Scroll View. They should be at the same level in the view hierarchy:



The Page Control should be a "sibling" of the Scroll View, not a child

If you did drop your Page Control inside the Scroll View instead of on top, then you can rearrange them inside the document pane.

That concludes the design of the landscape screen. The rest you will do in code, not in Interface Builder.



The final design of the landscape scene

Note: On macOS Sierra the scroll view doesn't say "UIScrollView" in the storyboard. It just appears as a transparent rectangle.

The other view controllers all employ Auto Layout to resize them to the dimensions of the user's device, but here you're going to take a different approach. Instead of pinning the Scroll View to the sides of the scene, you'll disable Auto Layout for this view controller and do the entire layout programmatically. So you don't need to pin anything in the storyboard.

You do need to hook up these controls to outlets, of course.

➤ Add these outlets to **LandscapeViewController.swift**, and connect them in Interface Builder:

```
@IBOutlet weak var scrollView: UIScrollView!
@IBOutlet weak var pageControl: UIPageControl!
```

Now to disable Auto Layout for this view controller. You can't use the storyboard's "Use Auto Layout" checkbox as it would turn off Auto Layout for all view controllers, not just this one.

➤ Replace **LandscapeViewController.swift**'s `viewDidLoad()` method with:

```
override func viewDidLoad() {
  super.viewDidLoad()

  view.removeConstraints(view.constraints)
  view.translatesAutoresizingMaskIntoConstraints = true

  pageControl.removeConstraints(pageControl.constraints)
  pageControl.translatesAutoresizingMaskIntoConstraints = true

  scrollView.removeConstraints(scrollView.constraints)
  scrollView.translatesAutoresizingMaskIntoConstraints = true
}
```

Remember how, if you don't add make constraints of your own, Interface Builder will give the views automatic constraints? Well, those automatic constraints get in the way if you're going to do your own layout. That's why you need to remove these unwanted constraints from the main `view`, `pageControl`, and `scrollView` first.

You also do `translatesAutoresizingMaskIntoConstraints = true`. That allows you to position and size your views manually by changing their `frame` property.

When Auto Layout is enabled, you're not really supposed to change the `frame` yourself – you can only indirectly move views into position by creating constraints. Modifying the `frame` by hand can cause conflicts with the existing constraints and bring all sorts of trouble (you don't want to make Auto Layout angry!).

For this view controller it's much more convenient to manipulate the `frame` property directly than it is making constraints (especially when you're placing the buttons for the search results), which is why you're disabling Auto Layout.

Note: Auto Layout doesn't really get disabled, but with the "translates autoresizing mask" option set to true, UIKit will convert your manual layout code into the proper constraints behind the scenes. That's also why you removed the automatic constraints because they will conflict with the new ones, causing your app to crash.

Now that Auto Layout is out of the way, you can do your own layout. That happens in the method `viewWillLayoutSubviews()`.

➤ Add this new method:

```
override func viewWillLayoutSubviews() {
  super.viewWillLayoutSubviews()

  scrollView.frame = view.bounds

  pageControl.frame = CGRect(
              x: 0,
              y: view.frame.size.height - pageControl.frame.size.height,
              width: view.frame.size.width,
              height: pageControl.frame.size.height)
}
```

The `viewWillLayoutSubviews()` method is called by UIKit as part of the layout phase of your view controller when it first appears on the screen. It's the ideal place for changing the `frame`s of your views by hand.

The scroll view should always be as large as the entire screen, so you make its `frame` equal to the main view's `bounds`.

The page control is located at the bottom of the screen, and spans the width of the screen. If this calculation doesn't make any sense to you, then try to sketch what happens on a piece of paper. It's what I usually do when writing my own layout code.

Note: Remember that the `bounds` describe the rectangle that makes up the inside of a view, while the `frame` describes the outside of the view.

The scroll view's `frame` is the rectangle seen from the perspective of the main view, while the scroll view's `bounds` is the same rectangle from the perspective of the scroll view itself.

Because the scroll view and page control are both children of the main view, their `frame`s sit in the same *coordinate space* as the `bounds` of the main view.

➤ Run the app and flip to landscape. Nothing much happens yet: the screen has the page control at the bottom (the dots) but it still mostly black.

For the scroll view to do anything you have to add some content to it.

➤ Add the following lines to `viewDidLoad()`:

```
scrollView.backgroundColor = UIColor(patternImage:
                                UIImage(named: "LandscapeBackground")!)
```

This puts an image on the scroll view's background so you can actually see something happening when you scroll through it.

An image? But you're setting the `backgroundColor` property, which is a `UIColor`, not a `UIImage`?! Yup, that's true, but `UIColor` has a cool trick that lets you use a tile-able image for a color.

If you take a peek at the **LandscapeBackground** image in the asset catalog you'll see that it is a small square. By setting this image as a pattern image on the background you get a repeatable image that fills the whole screen. You can use tile-able images anywhere you can use a `UIColor`.

Exercise: Why is the `!` needed behind the call to `UIImage(named: …)`? ■

Answer: `UIImage(named)` is a failable initializer and therefore returns an optional. Before you can use it as an actual `UIImage` object you need to unwrap it somehow. Here you know that the image will always exist so you can force unwrap with `!`.

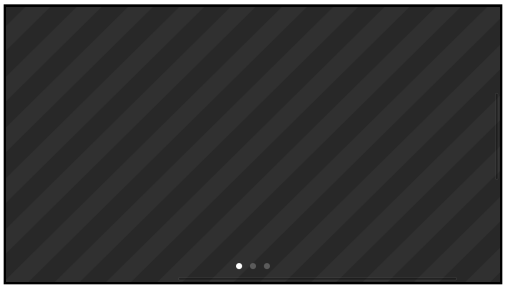
➤ Also add the following line to `viewDidLoad()`:

```
scrollView.contentSize = CGSize(width: 1000, height: 1000)
```

It is very important when dealing with scroll views that you set the `contentSize` property. This tells the scroll view how big its insides are. You don't change the `frame` (or `bounds`) of the scroll view if you want its insides to be bigger, you set the `contentSize` property instead.

People often forget this step and then they wonder why their scroll view doesn't scroll. Unfortunately, you cannot set `contentSize` from Interface Builder, so it must be done from code.

➤ Run the app and try some scrolling:



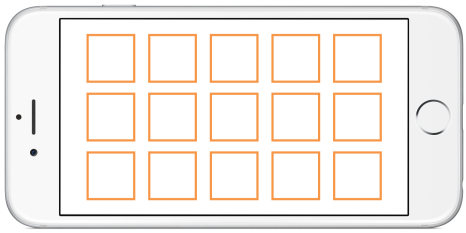
The scroll view now has a background image and it can scroll

If the dots at the bottom also move while scrolling, then you've placed the page control inside the scroll view. Open the storyboard and in the outline pane drag Page Control below Scroll View instead.

The page control itself doesn't do anything yet. Before you can make that work, you first have to add some content to the scroll view.

Adding buttons for the search results

The idea is to show the search results in a grid:



Each of these results is really a button. Before you can place these buttons on the screen, you need to calculate how many will fit on the screen at once. Easier said than done, because different iPhone models have different screen sizes.

Time for some math! Let's assume the app runs on a 3.5-inch device. In that case the scroll view is 480 points wide by 320 points tall. It can fit 3 rows of 5 columns if you put each search result in a rectangle of 96 by 88 points.

That comes to 3×5 = 15 search results on the screen at once. A search may return up to 200 results, so obviously there is not enough room for everything and you will have to spread out the results over several pages.

One page contains 15 buttons. For the maximum number of results you will need 200 / 15 = 13.3333 pages, which rounds up to 14 pages. That last page will only be filled for one-third with results.

The arithmetic for a 4-inch device is similar. Because the screen is wider – 568 instead of 480 points – it has room for an extra column, but only if you shrink each rectangle to 94 points instead of 96. That also leaves 568 – 94×6 = 4 points to spare.

The 4.7-inch iPhone 6s and 7 have room for 7 columns plus some leftover vertical space, and the 5.5-inch iPhone 6s Plus and 7 Plus can fit yet another column plus an extra row.

That's a lot of different possibilities!

You need to put all of this into an algorithm in `LandscapeViewController` so it can calculate how big the scroll view's `contentSize` has to be. It will also need to add a `UIButton` object for each search result.

Once you have that working, you can put the artwork image inside that `UIButton`.

Of course, this means the app first needs to give the array of search results to `LandscapeViewController` so it can use them for its calculations.

➤ Let's add a property for this, in **LandscapeViewController.swift**:

```
var searchResults = [SearchResult]()
```

Initially this has an empty array. `SearchViewController` gives it the real array upon rotation to landscape.

➤ Assign the array to the new property in **SearchViewController.swift**:

```
func showLandscape(with coordinator:
                              UIViewControllerTransitionCoordinator) {
  . . .
  if let controller = landscapeViewController {
    controller.searchResults = searchResults         // add this line
    controller.view.frame = view.bounds
    . . .
```

You have to be sure to fill up `searchResults` before you access the `view` property from the `LandscapeViewController`, because that will trigger the view to be loaded and performs `viewDidLoad()`.

The view controller will be reading from the `searchResults` array in `viewDidLoad()` to build up the contents of its scroll view. But if you access `controller.view` before setting `searchResults`, this property will still be `nil` and there is nothing to make buttons for. The order in which you do things matters!

➤ Switch back to **LandscapeViewController.swift**. From `viewDidLoad()` remove the line that sets `scrollView.contentSize`. That was just for testing.

Now let's go make those buttons.

➤ Add a new instance variable:

```
private var firstTime = true
```

The purpose for this variable will become clear in a moment. You need to initialize it with the value `true`.

Private parts

You are declaring the `firstTime` instance variable as `private`. You're doing this because `firstTime` is an internal piece of state that only `LandscapeViewController` cares about. It should not be visible to other objects.

You don't want the other objects in your app to know about the existence of `firstTime`, or worse, actually try to use this variable. Strange things are bound to happen if some other view controller changes the value of `firstTime` while `LandscapeViewController` doesn't expect that.

We haven't talked much about the distinction between *interface* and *implementation* yet, but what an object shows on the outside is different from what it has on the inside. That's done on purpose because its internals – the so-called implementation details – are not interesting to anyone else, and are often even dangerous to expose (messing around with them can crash the app).

It is considered good programming practice to hide as much as possible inside the object and only show a few things on the outside. The `firstTime` variable is only important to the insides of `LandscapeViewController`. Therefore it should not be part of its public interface, so that other objects can't see `firstTime` when they look at `LandscapeViewController`.

To make certain variables and methods visible only inside your own class, you declare them to be `private`. That removes them from the object's public interface.

Exercise: Find other variables and methods in the app that can be made `private`. ■

➤ Add the following lines to the bottom of `viewWillLayoutSubviews()`:

```
if firstTime {
  firstTime = false
  tileButtons(searchResults)
}
```

This calls a new method, `tileButtons()`, that performs the math and places the buttons on the screen in neat rows and columns. This needs to happen just once, when the `LandscapeViewController` is added to the screen.

You may think that `viewDidLoad()` would be a good place for that, but at the point in the view controller's lifecycle when `viewDidLoad()` is called, the view is not on the screen yet and has not been added into the view hierarchy. At this time it doesn't know how large it should be. Only after `viewDidLoad()` is done does the view get resized to fit the actual screen.

So you can't use `viewDidLoad()` for that. The only safe place to perform calculations based on the final size of the view – that is, any calculations that use the view's `frame` or `bounds` – is in `viewWillLayoutSubviews()`.

A warning: `viewWillLayoutSubviews()` may be invoked more than once! For example, it's also called when the landscape view gets removed from the screen again. You use the `firstTime` variable to make sure you only place the buttons once.

➤ Add the new `tileButtons()` method. It's a whopper, so we'll take it piece-by-piece.

```
private func tileButtons(_ searchResults: [SearchResult]) {
  var columnsPerPage = 5
  var rowsPerPage = 3
  var itemWidth: CGFloat = 96
  var itemHeight: CGFloat = 88
  var marginX: CGFloat = 0
  var marginY: CGFloat = 20

  let scrollViewWidth = scrollView.bounds.size.width

  switch scrollViewWidth {
  case 568:
    columnsPerPage = 6
    itemWidth = 94
    marginX = 2

  case 667:
    columnsPerPage = 7
    itemWidth = 95
    itemHeight = 98
    marginX = 1
    marginY = 29

  case 736:
    columnsPerPage = 8
    rowsPerPage = 4
    itemWidth = 92

  default:
    break
  }

  // TODO: more to come here
}
```

First, the method must decide on how big the grid squares will be and how many squares you need to fill up each page. There are four cases to consider, based on the width of the screen:

- **480 points**, 3.5-inch device (used when you run the app on an iPad). A single page fits 3 rows (`rowsPerPage`) of 5 columns (`columnsPerPage`). Each grid square is 96 by 88 points (`itemWidth` and `itemHeight`). The first row starts at Y = 20 (`marginY`).
- **568 points**, 4-inch device (all iPhone 5 models, iPhone SE). This has 3 rows of 6 columns. To make it fit, each grid square is now only 94 points wide. Because 568 doesn't evenly divide by 6, the `marginX` variable is used to adjust for the 4 points that are left over (2 on each side of the page).
- **667 points**, 4.7-inch device (iPhone 6, 6s, 7). This still has 3 rows but 7 columns. Because there's some extra vertical space, the rows are higher (98 points) and there is a larger margin at the top.
- **736 points**, 5.5-inch device (iPhone 6/6s/7 Plus). This device is huge and can house 4 rows of 8 columns.

The variables at the top of the method keep track of all these measurements.

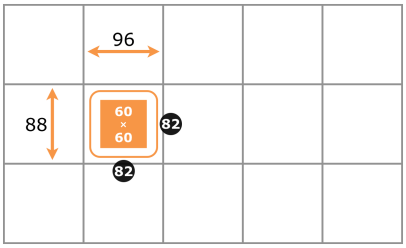
> **Note:** Wouldn't it be possible to come up with a nice formula that calculates all this stuff for you, rather than *hard-coding* these sizes and margin values? Probably, but it won't be easy. There are two things you want to optimize for: getting the maximum number of rows and columns on the screen, but at the same time not making the grid squares too small. Give it a shot if you think you can solve this puzzle! (Let me know if you do – I might put your solution in the next book update.)

➤ Add the following lines to `tileButtons()`:

```
let buttonWidth: CGFloat = 82
let buttonHeight: CGFloat = 82
let paddingHorz = (itemWidth - buttonWidth)/2
let paddingVert = (itemHeight - buttonHeight)/2
```

You've already determined that each search result gets a grid square of give-or-take 96 by 88 points (depending on the device), but that doesn't mean you need to make the buttons that big as well.

The image you'll put on the buttons is 60×60 pixels, so that leaves quite a gap around the image. After playing with the design a bit, I decided that the buttons will be 82×82 points (`buttonWidth` and `buttonHeight`), leaving a small amount of padding between each button and its neighbors (`paddingHorz` and `paddingVert`).



The dimensions of the buttons in the 5x3 grid

Now you can loop through the array of search results and make a new button for each `SearchResult` object.

➤ Add the following lines:

```
var row = 0
var column = 0
var x = marginX
for (index, searchResult) in searchResults.enumerated() {
  // 1
  let button = UIButton(type: .system)
  button.backgroundColor = UIColor.white
  button.setTitle("\(index)", for: .normal)
  // 2
  button.frame = CGRect(
                    x: x + paddingHorz,
                    y: marginY + CGFloat(row)*itemHeight + paddingVert,
                    width: buttonWidth, height: buttonHeight)
  // 3
  scrollView.addSubview(button)
  // 4
  row += 1
  if row == rowsPerPage {
    row = 0; x += itemWidth; column += 1

    if column == columnsPerPage {
      column = 0; x += marginX * 2
    }
  }
}
```

Here is how this works:

1. Create the `UIButton` object. For debugging purposes you give each button a title with the array index. If there are 200 results in the search, you also should end up with 200 buttons. Setting the index on the button will help to verify this.

2. When you make a button by hand you always have to set its `frame`. Using the measurements you figured out earlier, you determine the position and size of the button. Notice that `CGRect`'s fields all have the `CGFloat` type but `row` is an `Int`. You need to convert `row` to a `CGFloat` before you can use it in the calculation.

3. You add the new button object as a subview to the `UIScrollView`. After the first 18 or so buttons (depending on the screen size) this places any subsequent button out of the visible range of the scroll view, but that's the whole point. As long as you set the scroll view's `contentSize` accordingly, the user can scroll to get to those other buttons.

4. You use the `x` and `row` variables to position the buttons, going from top to bottom (by increasing `row`). When you've reached the bottom (`row` equals `rowsPerPage`), you go up again to row 0 and skip to the next column (by increasing the `column` variable).

When the `column` reaches the end of the screen (equals `columnsPerPage`), you reset it to 0 and add any leftover space to `x` (twice the X-margin). This only has an effect on 4-inch and 4.7-inch screens; for the others `marginX` is 0.

Note that in Swift you can put multiple statements on a single line by separating them with a semicolon. I did that to save some space.

If this sounds like hocus pocus to you, I suggest you play around a bit with these calculations to gain insight into how they work. It's not rocket science but it does require some mental gymnastics. Tip: Sketching the process on paper can help!

> **Note:** By the way, did you notice what happened in the `for in` loop?
>
> `for (index, searchResult) in searchResults.enumerated() {`
>
> This `for in` loop steps through the `SearchResult` objects from the array, but with a twist. By doing `for in enumerated()`, you get a *tuple* containing not only the next `SearchResult` object but also its index in the array.
>
> A tuple is nothing more than a temporary list with two or more items in it. Here, the tuple is `(index, searchResult)`. This is a neat trick to loop through an array and get both the objects and their indices.

➤ Finally, add the last part of this very long method:

```
let buttonsPerPage = columnsPerPage * rowsPerPage
let numPages = 1 + (searchResults.count - 1) / buttonsPerPage

scrollView.contentSize = CGSize(
                             width: CGFloat(numPages)*scrollViewWidth,
                             height: scrollView.bounds.size.height)

print("Number of pages: \(numPages)")
```

At the end of the method you calculate the `contentSize` for the scroll view based on how many buttons fit on a page and the number of `SearchResult` objects.

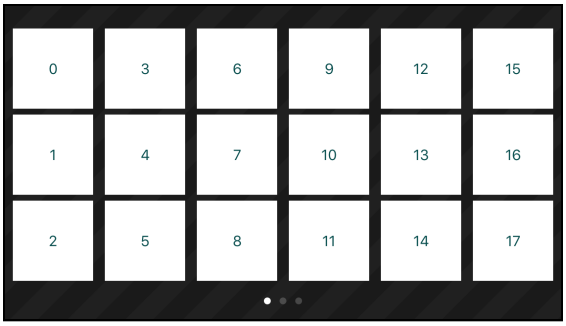
You want the user to be able to "page" through these results, rather than simply scroll (a feature that you'll enable shortly) so you should always make the content width a multiple of the screen width (480, 568, 667 or 736 points).

With a simple formula you can then determine how many pages you need.

> **Note:** Dividing an integer value by an integer always results in an integer. If `buttonsPerPage` is 18 (3 rows × 6 columns) and there are fewer than 18 search results, `searchResults.count / buttonsPerPage` is 0.
>
> It's important to realize that `numPages` will never have a fractional value because all the variables involved in the calculation are `Int`s, which makes `numPages` an `Int` too.
>
> That's why the formula is `1 + (searchResults.count – 1) / buttonsPerPage`.
>
> If there are 18 results, exactly enough to fill a single page, `numPages = 1 + 17/18 = 1 + 0 = 1`. But if there are 19 results, the 19th result needs to go on the second page, and `numPages = 1 + 18/18 = 1 + 1 = 2`. Plug in some other values for yourself to prove this formula is correct.

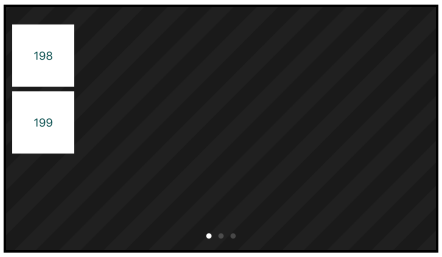
I also threw in a `print()` for good measure, so you can verify that you really end up with the right amount of pages.

➤ Run the app, do a search, and flip to landscape. You should now see a whole bunch of buttons:



The landscape view has buttons

Scroll all the way to the right and it looks like this (on the iPhone SE):



The last page of the search results

That is 200 buttons indeed (you started counting at 0, remember?).

Just to make sure that this logic works properly you should test a few different scenarios. What happens when there are fewer results than 18 (the amount that fit on a single page on the iPhone 5)? What happens when there are exactly 18 search results? How about 19, one more than can go on a single page?

The easiest way to create this situation is to change the `&limit` parameter in the search URL.

Exercise. Try these situations for yourself and see what happens. ■

➤ Also test when there are no search results. The landscape view should now be empty. In a short while you'll add a "Nothing Found" label to this screen as well.

> **Note:** Xcode currently gives a warning "Immutable value searchResult was never used; consider replacing with _". That warning will go away once you use the `searchResult` variable in the next section.

Paging

So far the Page Control at the bottom of the screen has always shown three dots. And there wasn't much paging to be done on the scroll view either.

In case you're wondering what "paging" means: if the user has dragged the scroll view a certain amount, it should snap to a new page.

With paging enabled, you can quickly flick through the contents of a scroll view, without having to drag it all the way. You're no doubt familiar with this effect because it is what the iPhone uses on its springboard. Many other apps use the effect too, such as the Weather app that uses paging to flip between the cards for different cities.

➤ Go to the **LandscapeViewController** in the storyboard and check the **Paging Enabled** option for the scroll view (in the Attributes inspector).

There, that was easy. Now run the app and the scroll view will let you page rather than scroll. That's cool but you also need to do something with the page control at the bottom.

➤ Add this line to `viewDidLoad()`:

```
pageControl.numberOfPages = 0
```

This effectively hides the page control, which is what you want to do when there are no search results (yet).

➤ Add the following lines to the bottom of `tileButtons()`:

```
pageControl.numberOfPages = numPages
pageControl.currentPage = 0
```

This sets the number of dots that the page control displays to the number of pages that you calculated.

The active dot (the white one) needs to be synchronized with the active page in the scroll view. Currently, it never changes unless you tap in the page control and even then it has no effect on the scroll view.

To get this to work, you'll have to make the page control talk to the scroll view, and vice versa. The view controller must become the delegate of the scroll view so it will be notified when the user is flicking through the pages.

➤ Add the following to the very bottom of **LandscapeViewController.swift**:

```
extension LandscapeViewController: UIScrollViewDelegate {
  func scrollViewDidScroll(_ scrollView: UIScrollView) {
    let width = scrollView.bounds.size.width
    let currentPage = Int((scrollView.contentOffset.x + width/2)/width)
    pageControl.currentPage = currentPage
  }
}
```

This is one of the `UIScrollViewDelegate` methods. You figure out what the index of the current page is by looking at the `contentOffset` property of the scroll view. This property determines how far the scroll view has been scrolled and is updated while you're dragging the scroll view.

Unfortunately, the scroll view doesn't simply tell us, "The user has flipped to page X", and so you have to calculate this yourself. If the content offset gets beyond halfway on the page `(width/2)`, the scroll view will flick to the next page. In that case, you update the `pageControl`'s active page number.

You also need to know when the user taps on the Page Control so you can update the scroll view. There is no delegate for this but you can use a regular `@IBAction` method.

➤ Add the action method:

```
@IBAction func pageChanged(_ sender: UIPageControl) {
  scrollView.contentOffset = CGPoint(
    x: scrollView.bounds.size.width * CGFloat(sender.currentPage), y: 0)
}
```

This works the other way around: when the user taps in the Page Control, its `currentPage` property gets updated. You use that to calculate a new `contentOffset` for the scroll view.

➤ In the storyboard, **Ctrl-drag** from the Scroll View to Landscape View Controller and select **delegate**.

➤ Also **Ctrl-drag** from the Page Control to the Landscape View Controller and select **pageChanged:** under Sent Events.

➤ Try it out, the page control and the scroll view should now be in sync.

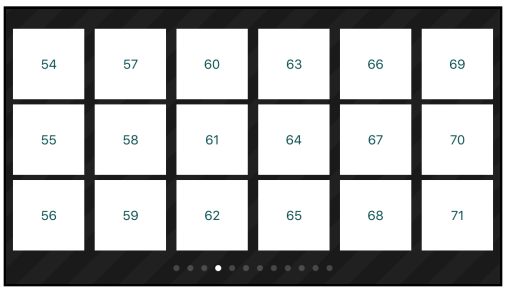
The transition from one page to another after tapping in the page control is still a little abrupt, though. An animation would help here.

Exercise. See if you can animate what happens in `pageChanged()`. ■

Answer: You can simply put the above code in an animation block:

```
@IBAction func pageChanged(_ sender: UIPageControl) {
  UIView.animate(withDuration: 0.3, delay: 0,
                 options: [.curveEaseInOut], animations: {
   self.scrollView.contentOffset = CGPoint(
     x: self.scrollView.bounds.size.width * CGFloat(sender.currentPage),
     y: 0)
  },
  completion: nil)
}
```

You're using a version of the `UIView` animation method that allows you to specify options because the "Ease In, Ease Out" timing (`.curveEaseInOut`) looks good here.



We've got paging!

➤ This is a good time to commit.

Downloading artwork on the buttons

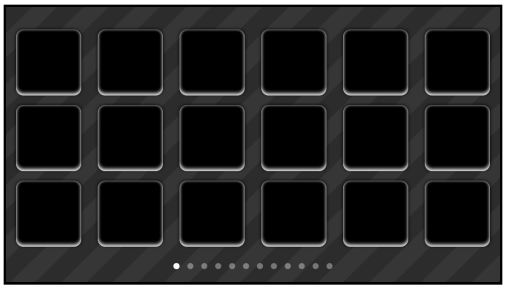
First let's give the buttons a nicer look.

➤ Replace the button creation code in `tileButtons()` with:

```
let button = UIButton(type: .custom)
button.setBackgroundImage(UIImage(named: "LandscapeButton"),
                          for: .normal)
```

Instead of a regular button you're now making a `.custom` one, and you're giving it a background image instead of a title.

If you run the app, it will look like this:



The buttons now have a custom background image

Now you will have to download the artwork images (if they haven't been already downloaded and cached yet by the table view) and put them on the buttons.

Problem: You're dealing with `UIButton`s here, not `UIImageView`s, so you cannot simply use that handy extension from earlier. Fortunately, the code is very similar!

➤ Add a new method to **LandscapeViewController.swift**:

```
private func downloadImage(for searchResult: SearchResult,
                           andPlaceOn button: UIButton) {
  if let url = URL(string: searchResult.artworkSmallURL) {
    let downloadTask = URLSession.shared.downloadTask(with: url) {
      [weak button] url, response, error in

      if error == nil, let url = url,
                       let data = try? Data(contentsOf: url),
                       let image = UIImage(data: data) {
        DispatchQueue.main.async {
          if let button = button {
            button.setImage(image, for: .normal)
          }
        }
      }
    }
    downloadTask.resume()
  }
}
```

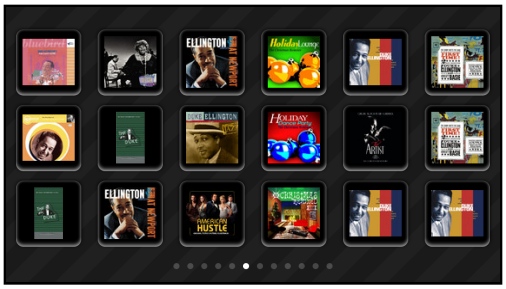
This looks very much like what you did in the `UIImageView` extension.

First you get a `URL` object with the link to the 60×60-pixel artwork, and then you create a download task. Inside the completion handler you put the downloaded file into a `UIImage`, and if all that succeeds, use `DispatchQueue.main.async` to place the image on the button.

➤ Add the following to `tileButtons()` to call this new method, right after you create the button:

```
downloadImage(for: searchResult, andPlaceOn: button)
```

And that should do it. Run the app and you'll get some cool-looking buttons:



Showing the artwork on the buttons

> **Note:** The Xcode warning about `searchResult` is gone, but now it gives the same message for the `index` variable. Xcode doesn't like it if you declare variables but not use them. You'll use `index` again later in this tutorial but in the mean time you can replace it by the _ wildcard symbol to stop Xcode from complaining.

It's always a good idea to clean up after yourself, also in programming. Imagine this: what would happen if the app is still downloading images and the user flips back to portrait mode?

At that point, the `LandscapeViewController` is deallocated but the image downloads keep going. That is exactly the sort of situation that can crash your app if you don't handle it properly.

To avoid ownership cycles, you capture the button with a `weak` reference. When `LandscapeViewController` is deallocated, so are the buttons and the completion handler's captured `button` reference automatically becomes `nil`. The `if let` inside the `DispatchQueue.main.async` block will now safely skip `button.setImage(for)`. No harm done. That's why you wrote `[weak button]`.

However, to conserve resources the app should really stop downloading these images because they end up nowhere. Otherwise it's just wasting bandwidth and battery life, and users don't take too kindly to apps that do.

➤ Add a new instance variable to **LandscapeViewController.swift**:

```
private var downloadTasks = [URLSessionDownloadTask]()
```

This array keeps track of all the active `URLSessionDownloadTask` objects.

➤ Add the following line to the bottom of `downloadImage(for:andPlaceOn:)`, right after where you resume the download task:

```
downloadTasks.append(downloadTask)
```

➤ And finally, tell `deinit` to cancel any operations that are still on the way:

```
deinit {
  print("deinit \(self)")
  for task in downloadTasks {
    task.cancel()
  }
}
```

This will stop the download for any button whose image was still pending or in transit. Good job, partner!

➤ Commit your changes.

Exercise. Despite what the iTunes web service promises, not all of the artwork is truly 60×60 pixels. Some of it is bigger, some is not even square, and so it might not always fit nicely in the button. Your challenge is to use the image sizing code from MyLocations to always resize the image to 60×60 points before you put it on the button. Note that we're talking points here, not pixels – on Retina devices the image should actually end up being 120×120 or even 180×180 pixels big. ■

You can find the project files for the app up to this point under **07 - Landscape** in the tutorial's Source Code folder.

> **Note:** In this section you learned how to create a grid-like view using a `UIScrollView`. iOS comes with a versatile class, `UICollectionView`, that lets you do the same thing – and much more! – without having to resort to the sort of math you did in `tileButtons()`. To learn more about `UICollectionView`, check out the website: raywenderlich.com/tag/collection-view

Refactoring the search

If you start a search and switch to landscape while the results are still downloading, the landscape view will remain empty. It would also be nice to show an activity spinner on that screen while the search is taking place.

You can reproduce this situation by artificially slowing down your network connection using the Network Link Conditioner tool.

So how can `LandscapeViewController` tell what state the search is in? Its `searchResults` array will be empty if no search was done yet and have zero or more `SearchResult` objects after a successful search.

Just by looking at the array object you cannot determine whether the search is still going, or whether it has finished but nothing was found. In both cases, the `searchResults` array will have a count of 0.

You need a way to determine whether the search is still busy. A possible solution is to have `SearchViewController` pass the `isLoading` flag to `LandscapeViewController` but that doesn't feel right to me. This is known as a "code smell", a hint at a deeper problem with the design of the program.

Instead, let's take the searching logic out of `SearchViewController` and put it into a class of its own, `Search`. Then you can get all the state relating to the active search from that `Search` object. Time for some more refactoring!

➤ If you want, create a new branch for this in Git.

This is a pretty invasive change in the code and there is always a risk that it doesn't work as well as you hoped. By making the changes in a new branch, you can commit every once in a while without messing up the master branch. Making new branches in Git is quick and easy, so it's good to get into the habit.

➤ Create a new file using the **Swift File** template. Name it **Search**.

➤ Change the contents of **Search.swift** to:

```
import Foundation

class Search {
  var searchResults: [SearchResult] = []
  var hasSearched = false
  var isLoading = false

  private var dataTask: URLSessionDataTask? = nil

  func performSearch(for text: String, category: Int) {
    print("Searching...")
  }
}
```

You've given this class three public properties, one private property, and a method. This stuff should look familiar because it comes straight from `SearchViewController`. You'll be removing code from that class and putting it into this new `Search` class.

The `performSearch(for:category:)` method doesn't do much yet but that's OK. First I want to make `SearchViewController` work with this new `Search` object and when that compiles without errors, you will move all the logic over. Small steps!

Let's make the changes to **SearchViewController.swift**. Xcode will probably give a bunch of errors and warnings while you're making these changes, but it will all work out in the end.

➤ In **SearchViewController.swift**, remove the declarations for the following instance variables:

```
var searchResults: [SearchResult] = []
var hasSearched = false
var isLoading = false
var dataTask: URLSessionDataTask?
```

and replace them with this one:

```
let search = Search()
```

The new `Search` object not only describes the state and results of the search, it also will have all the logic for talking to the iTunes web service. You can now remove a lot of code from the view controller.

➤ Cut the following methods and paste them into **Search.swift**:

- `iTunesURL(searchText, category)`
- `parse(json)`
- `parse(dictionary)`
- `parse(track)`
- `parse(audiobook)`
- `parse(software)`
- `parse(ebook)`

➤ Make these methods `private`. They are only important to `Search` itself, not to any other classes from the app, so it's good to "hide" them.

➤ Back in **SearchViewController.swift**, replace the `performSearch()` method with the following (tip: set aside the old code in a temporary file because you'll need it again later).

```
func performSearch() {
  search.performSearch(for: searchBar.text!,
                       category: segmentedControl.selectedSegmentIndex)

  tableView.reloadData()
  searchBar.resignFirstResponder()
}
```

This simply makes the `Search` object do all the work. Of course it still reloads the table view (to show the activity spinner) and hides the keyboard.

There are a few places in the code that still use the old `searchResults` array even though that no longer exists. You should change them to use the `searchResults` property from the `Search` object instead. Likewise for `hasSearched` and `isLoading`.

➤ For example, change `tableView(numberOfRowsInSection)` to:

```
func tableView(_ tableView: UITableView,
               numberOfRowsInSection section: Int) -> Int {
  if search.isLoading {
    return 1  // Loading...
  } else if !search.hasSearched {
    return 0  // Not searched yet
  } else if search.searchResults.count == 0 {
    return 1  // Nothing Found
  } else {
    return search.searchResults.count
  }
}
```

➤ In `showLandscape(with)`, change the line that sets the `searchResults` property on the new view controller to:

```
controller.search = search
```

This line still gives an error even after you've changed it but you'll fix that soon.

➤ Anywhere else in the code that says `isLoading` or `searchResults`, replace that with `search.isLoading` and `search.searchResults`.

The `LandscapeViewController` still has a property for a `searchResults` array so you have to change that to use the `Search` object as well.

➤ In **LandscapeViewController.swift**, remove the `searchResults` instance variable and replace it with:

```
var search: Search!
```

➤ In `viewWillLayoutSubviews()`, change the call to `tileButtons()` into:

```
tileButtons(search.searchResults)
```

OK, that's the first round of changes. Build the app to make sure there are no compiler errors.

The app itself doesn't do much anymore because you removed all the searching logic. So let's put that back in.

➤ In **Search.swift**, replace `performSearch(for:category:)` with the following (use that temporary file but be careful to make the proper changes):

```
func performSearch(for text: String, category: Int) {
  if !text.isEmpty {
```

```
      dataTask?.cancel()

      isLoading = true
      hasSearched = true
      searchResults = []

      let url = iTunesURL(searchText: text, category: category)

      let session = URLSession.shared
      dataTask = session.dataTask(with: url, completionHandler: {
        data, response, error in

        if let error = error as? NSError, error.code == -999 {
          return   // Search was cancelled
        }

        if let httpResponse = response as? HTTPURLResponse,
               httpResponse.statusCode == 200,
           let jsonData = data,
           let jsonDictionary = self.parse(json: jsonData) {

          self.searchResults = self.parse(dictionary: jsonDictionary)
          self.searchResults.sort(by: <)

          print("Success!")
          self.isLoading = false
          return
        }

        print("Failure! \(response)")
        self.hasSearched = false
        self.isLoading = false
      })
      dataTask?.resume()
    }
  }
```

This is basically the same thing you did before, except all the user interface logic has been removed. The purpose of `Search` is just to perform a search, it should not do any UI stuff. That's the job of the view controller.

➤ Run the app and search for something. When the search finishes, the debug pane shows a "Success!" message but the table view does not reload and the spinner keeps spinning in eternity.

The `Search` object currently has no way to tell the `SearchViewController` that it is done. You could solve this by making `SearchViewController` a delegate of the `Search` object, but for situations like these closures are much more convenient.

So let's create your own closure!

➤ Add the following line to **Search.swift**, above the `class` line:

```
  typealias SearchComplete = (Bool) -> Void
```

The `typealias` statement allows you to create a more convenient name for a data type, in order to save some keystrokes and to make the code more readable.

Here you're declaring a type for your own closure, named `SearchComplete`. This is a closure that returns no value (it is `Void`) and takes one parameter, a `Bool`. If you think this syntax is weird, then I'm right there with you, but that's the way it is.

From now on you can use the name `SearchComplete` to refer to a closure that takes one `Bool` parameter and returns no value.

★ ★ ★

Closure types

Whenever you see a `->` in a type definition, the type is intended for a closure, function, or method.

Swift treats these three things as mostly interchangeable. Closures, functions, and methods are all blocks of source code that possibly take parameters and return a value. The difference is that a function is really just a closure with a name, and a method is a function that lives inside an object.

Some examples of closure types:

`() -> ()` is a closure that takes no parameters and returns no value.

`Void -> Void` is the same as the previous example. `Void` and `()` mean the same thing.

`(Int) -> Bool` is a closure that takes one parameter, an `Int`, and returns a `Bool`.

`Int -> Bool` is this is the same as above. If there is only one parameter, you can leave out the parentheses.

`(Int, String) -> Bool` is a closure taking two parameters, an `Int` and a `String`, and returning a `Bool`.

`(Int, String) -> Bool?` as above but now returns an optional `Bool` value.

`(Int) -> (Int) -> Int` is a closure that returns another closure that returns an `Int`. Freaky! Swift treats closures like any other type of object, so you can also pass them as parameters and return them from functions.

★ ★ ★

➤ Make the following changes to `performSearch(for:category:)`:

```
func performSearch(for text: String, category: Int,
                   completion: @escaping SearchComplete) {      // new
  if !text.isEmpty {
    . . .
    dataTask = session.dataTask(with: url, completionHandler: {
      data, response, error in

      var success = false                                       // new

      if let error = error . . . {
        return   // Search was cancelled
      }
      if let httpResponse = response as? . . . {
        . . .
        self.isLoading = false
        success = true                              // instead of return
      }

      if !success {                                             // new
        self.hasSearched = false
        self.isLoading = false
      }

      DispatchQueue.main.async {                                // new
        completion(success)
      }
    })
    dataTask?.resume()
  }
}
```

You've added a third parameter named `completion` that is of type `SearchComplete`. Whoever calls `performSearch(for, category, completion)` can now supply their own closure, and the method will execute the code that is inside that closure when the search completes.

> **Note:** The `@escaping` annotation is necessary for closures that are not used immediately. It tells Swift that this closure may need to capture variables such as `self` and keep them around for a little while until the closure can finally be executed (when the search is done).

Instead of returning early from the closure upon success, you now set the `success` variable to `true` (this replaces the `return` statement). The value of `success` is used for the `Bool` parameter of the `completion` closure, as you can see inside the call to `DispatchQueue.main.async` at the bottom.

To perform the code from the closure, you simply call it as you'd call any function or method: `closureName(parameters)`. You call `completion(true)` upon success and `completion(false)` upon failure. This is done so that the `SearchViewController` can reload its table view or, in the case of an error, show an alert view.

➤ In **SearchViewController.swift**, replace `performSearch()` with:

```
func performSearch() {
  search.performSearch(for: searchBar.text!,
                       category: segmentedControl.selectedSegmentIndex,
                       completion: { success in
    if !success {
      self.showNetworkError()
    }
    self.tableView.reloadData()
  })

  tableView.reloadData()
  searchBar.resignFirstResponder()
}
```

You now pass a closure to `performSearch(for, category, completion)`. The code in this closure gets called after the search completes, with the `success` parameter being either `true` or `false`. A lot simpler than making a delegate, no? The closure is always called on the main thread, so it's safe to use UI code here.

➤ Run the app. You should be able to search again.

That's the first part of this refactoring complete. You've extracted the relevant code for searching out of the `SearchViewController` and placed it into its own object, `Search`. The view controller now only does view-related things, which is exactly what it is supposed to do and no more.

➤ You've made quite a few extensive changes, so it's a good idea to commit.

Improving the categories

The idea behind Swift's strong typing is that the data type of a variable should be as descriptive as possible. Right now the category to search for is represented by a number, 0 to 3, but is that the best way to describe a category to your program?

If you see the number 3 does that mean "e-book" to you? It could be anything… And what if you use 4 or 99 or -1, what would that mean? These are all valid values for an `Int` but not for a category. The only reason the category is currently an `Int` is because `segmentedControl.selectedSegmentIndex` is an `Int`.

There are only four possible search categories, so this sounds like an excellent job for an enum.

➤ Add the following to **Search.swift**, *inside* the `class` brackets:

```
enum Category: Int {
  case all = 0
  case music = 1
  case software = 2
  case ebooks = 3
}
```

This creates a new enumeration type named `Category` with four possible items. Each of these has a numeric value associated with it, called the **raw** value.

Contrast this with the `AnimationStyle` enum you made before:

```
enum AnimationStyle {
  case slide
  case fade
}
```

This enum does not give numbers to its values (it also doesn't say "`: Int`" behind the enum name). For `AnimationStyle` it doesn't matter that `slide` is really number 0 and `fade` is number 1, or whatever the values might be. All you care about is that a variable of type `AnimationStyle` can either be `.slide` or `.fade` – a numeric value is not important.

For the `Category` enum, however, you want to connect its four items to the four possible indices of the Segmented Control. If segment 3 is selected, you want this to correspond to `.ebooks`. That's why the items from the `Category` enum do have numbers.

➤ Change the method signature of `performSearch(for, category, completion)` to use this new type:

```
func performSearch(for text: String, category: Category,
                   completion: @escaping SearchComplete) {
```

The `category` parameter is no longer an `Int`. It is not possible anymore to pass it the value 4 or 99 or -1. It must always be one of the values from the `Category` enum. This reduces a potential source of bugs and it has made the program more expressive. Whenever you have a limited list of possible values that can be turned into an enum, it's worth doing!

➤ Also change `iTunesURL(searchText, category)` because that also assumed `category` would be an `Int`:

```
private func iTunesURL(searchText: String, category: Category) -> URL {
  let entityName: String
  switch category {
  case .all: entityName = ""
  case .music: entityName = "musicTrack"
  case .software: entityName = "software"
  case .ebooks: entityName = "ebook"
  }

  let escapedSearchText = . . .
```

The `switch` now looks at the various cases from the `Category` enum instead of the numbers 0 to 3. Note that the `default` case is no longer needed because this enum cannot have any other values.

This code works, but to be honest I'm not entirely happy with it. I've said before that any logic that is related to an object should be an integral part of that object – in other words, an object should do as much as it can itself.

Converting the category into an "entity name" string that goes into the iTunes URL is a good example – that sounds like something the `Category` enum itself could do.

Swift enums can have their own methods and properties, so let's take advantage of that and improve the code even more.

➤ Add the `entityName` property to the `Category` enum:

```
enum Category: Int {
  case all = 0
  case music = 1
  case software = 2
  case ebooks = 3

  var entityName: String {
    switch self {
    case .all: return ""
    case .music: return "musicTrack"
    case .software: return "software"
    case .ebooks: return "ebook"
    }
  }
}
```

Swift enums cannot have instance variables, only computed properties. `entityName` has the exact same `switch` statement that you just saw, except that it switches on `self`, the current value of the enumeration object.

➤ In `iTunesURL(searchText, category)` you can now simply write:

```
private func iTunesURL(searchText: String, category: Category) -> URL {
  let entityName = category.entityName
  let escapedSearchText = . . .
```

That's a lot cleaner. Everything that has to do with categories now lives inside its own enum, `Category`.

You still need to tell `SearchViewController` about this, because it needs to convert the selected segment index into a proper `Category` value.

➤ In **SearchViewController.swift**, change the first part of `performSearch()` to:

```
func performSearch() {
  if let category = Search.Category(
                      rawValue: segmentedControl.selectedSegmentIndex) {
    search.performSearch(for: searchBar.text!, category: category,
                         completion: {
      . . .
    })
```

```
      tableView.reloadData()
      searchBar.resignFirstResponder()
    }
  }
```

To convert the `Int` value from `selectedSegmentIndex` to an item from the `Category` enum you use the built-in `init(rawValue)` method. This may fail, for example when you pass in a number that isn't covered by one of `Category`'s cases, i.e. anything that is outside the range 0 to 3. That's why `init(rawValue)` returns an optional that needs to be unwrapped with `if let` before you can use it.

> **Note:** Because you placed the `Category` enum inside the `Search` class, its full name is `Search.Category`. In other words, `Category` lives inside the `Search` *namespace*. It makes sense to bundle up these two things because they are so closely related.

➤ Build and run to see if the different categories still work. Nice!

Enums with associated values

Enums are pretty useful to restrict something to a limited range of possibilities, like what you did with the search categories. But they are even more powerful than you might have expected, as you'll find out in this section…

Like all objects, the `Search` object has a certain amount of *state*. For `Search` this is determined by its `isLoading`, `hasSearched`, and `searchResults` variables.

These three variables describe four possible states:

State	hasSearched	isLoading	searchResults
No search has been performed yet (this is also the state after an error)	false	false	Empty array
The search is in progress	true	true	Empty array
No results were found	true	false	Empty array
There are search results	true	false	Contains at least one `SearchResult` object

The `Search` object is in only one of these states at a time, and when it changes from one state to another there is a corresponding change in the app's UI. For example, upon a change from "searching" to "have results", the app hides the activity spinner and loads the results into the table view.

The problem is that this state is scattered across three different variables. It's tricky to see what the current state is just by looking at these variables (you may have to refer to the above table).

You can do better than that by giving `Search` an explicit `state` variable. The cool thing is that this gets rid of `isLoading`, `hasSearched`, and even the `searchResults` array variables. Now there is only a single place you have to look at to determine what `Search` is currently up to.

➤ In **Search.swift**, remove the following instance variables:

```
var searchResults: [SearchResult] = []
var hasSearched = false
var isLoading = false
```

➤ In their place, add the following enum (this goes inside the class again):

```
enum State {
  case notSearchedYet
  case loading
  case noResults
  case results([SearchResult])
}
```

This enumeration has a case for each of the four states listed above. It does not need raw values so the cases don't have numbers. (It's important to note that the state `.notSearchedYet` is also used for when there is an error.)

The `.results` case is special: it has a so-called **associated value**, which is an array of `SearchResult` objects.

This array is only important when the search was successful. In all the other cases, there are no search results and the array was empty anyway (see the above table). By making it an associated value, you'll only have access to this array when `Search` is in the `.results` state. In the other states, the array simply does not exist.

Let's see how this works.

➤ First add a new instance variable:

```
private(set) var state: State = .notSearchedYet
```

This keeps track of `Search`'s current state. Its initial value is `.notSearchedYet` – obviously no search has happened yet when the `Search` object is first constructed.

This variable is `private`, but only half. It's not unreasonable for other objects to want to ask `Search` what its current state is. In fact, the app won't work unless you allow this.

But you don't want those other objects to be able to *change* the value of `state`; they are only allowed to read the state value. With `private(set)` you tell Swift that

reading is OK for other objects, but assigning new values to this variable may only happen inside the `Search` class.

➤ Change `performSearch(for, category, completion)` to use this new variable:

```
func performSearch(for text: String, category: Category,
                   completion: @escaping SearchComplete) {
  if !text.isEmpty {
    dataTask?.cancel()

    state = .loading                                          // add this

    let url = iTunesURL(searchText: text, category: category)

    let session = URLSession.shared
    dataTask = session.dataTask(with: url, completionHandler: {
      data, response, error in

      var newState = State.notSearchedYet                     // add this
      var success = false

      if let error = error as? NSError, error.code == -999 {
        return   // Search was cancelled
      }

      if let httpResponse = response as? HTTPURLResponse,
        httpResponse.statusCode == 200,
        let jsonData = data,
        let jsonDictionary = self.parse(json: jsonData) {

        // change this entire section
        var searchResults = self.parse(dictionary: jsonDictionary)
        if searchResults.isEmpty {
          newState = .noResults
        } else {
          searchResults.sort(by: <)
          newState = .results(searchResults)
        }
        success = true
      }

      DispatchQueue.main.async {
        self.state = newState                                 // add this
        completion(success)
      }
    })
    dataTask?.resume()
  }
}
```

Instead of the old variables `isLoading`, `hasSearched`, and `searchResults`, this now only changes `state`.

Note: You don't update `state` directly but instead use a new local variable `newState`. Then at the end, in the `DispatchQueue.main.async` block, you put the

value of `newState` back into `self.state`. The reason for doing this the long way round is that `state` must only be changed by the main thread, or it can lead to a nasty and unpredictable bug known as a *race condition*.

When you have multiple threads trying to use the same variable at the same time, the app may do unexpected things and crash. In our app, the main thread will try to use `search.state` to display the activity spinner in the table view – and that can happen at the same time as `URLSession`'s completion handler, which runs in a background thread. We have to make sure these two threads don't get in each other's way!

Here's how the new logic works. A lot can go wrong between performing the network request and parsing the JSON. By setting `newState` to `.notSearchedYet` (which doubles as the error state) and `success` to `false` at the start of the completion handler you assume the worst – always a good idea when doing network programming – unless there's evidence otherwise.

That evidence comes when the app was able to successfully parse the JSON and create an array of `SearchResult` objects. If the array is empty, `newState` becomes `.noResults`.

The interesting thing happens when the array is *not* empty. After sorting it like before, you do `newState = .results(searchResults)`. This gives `newState` the value `.results` and also associates the array of `SearchResult` objects with it. You no longer need a separate instance variable to keep track of the array; the array object is intrinsically attached to the value of `newState`.

Finally, you copy the value of `newState` into `self.state`. As I mentioned, this needs to happen on the main thread to prevent race conditions, which is why it is done in inside the `DispatchQueue.main.async` block.

That completes the changes in **Search.swift**, but there are quite a few other places in the code that still try to use `Search`'s old instance variables.

➤ In **SearchViewController.swift**, change `tableView(numberOfRowsInSection)` to:

```
func tableView(_ tableView: UITableView,
               numberOfRowsInSection section: Int) -> Int {
  switch search.state {
  case .notSearchedYet:
    return 0
  case .loading:
    return 1
  case .noResults:
    return 1
  case .results(let list):
    return list.count
  }
}
```

This is pretty straightforward. Instead of trying to make sense out of the separate `isLoading`, `hasSearched`, and `searchResults` variables, this simply looks at the value from `search.state`. The `switch` statement is ideal for situations like this.

The `.results` case requires more explanation. Because `.results` has an array of `SearchResult` objects associated with it, you can *bind* this array to a temporary variable, `list`, and then use that variable inside the case to read how many items are in the array. That's how you make use of the associated value.

This pattern, using a `switch` statement to look at `state`, is going to become very common in your code.

➤ Change `tableView(cellForRowAt)` to the following:

```
func tableView(_ tableView: UITableView,
               cellForRowAt indexPath: IndexPath) -> UITableViewCell {
  switch search.state {
  case .notSearchedYet:
    fatalError("Should never get here")

  case .loading:
    let cell = tableView.dequeueReusableCell(
                   withIdentifier: TableViewCellIdentifiers.loadingCell,
                   for: indexPath)

    let spinner = cell.viewWithTag(100) as! UIActivityIndicatorView
    spinner.startAnimating()
    return cell

  case .noResults:
    return tableView.dequeueReusableCell(
              withIdentifier: TableViewCellIdentifiers.nothingFoundCell,
              for: indexPath)

  case .results(let list):
    let cell = tableView.dequeueReusableCell(
              withIdentifier: TableViewCellIdentifiers.searchResultCell,
              for: indexPath) as! SearchResultCell

    let searchResult = list[indexPath.row]
    cell.configure(for: searchResult)
    return cell
  }
}
```

The same thing happened here. The various `if` statements have been replaced by a `switch` and `case` statements for the four possibilities.

Note that "`numberOfRowsInSection`" returns 0 for `.notSearchedYet` and no cells will ever be asked for. But because a switch must always be exhaustive, you also have to include a case for `.notSearchedYet` in "`cellForRowAt`". Considering that it's a bug when the code gets there you can use the built-in `fatalError()` function to help catch such mistakes.

➤ Next up is `tableView(willSelectRowAt):`

```
func tableView(_ tableView: UITableView,
               willSelectRowAt indexPath: IndexPath) -> IndexPath? {
  switch search.state {
  case .notSearchedYet, .loading, .noResults:
    return nil
  case .results:
    return indexPath
  }
}
```

It's only possible to tap on rows when the state is `.results`, so in all other cases this method returns `nil`. (You don't need to bind the results array because you're not using it for anything.)

➤ And finally, `prepare(for:sender:)`. Change it to:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
  if segue.identifier == "ShowDetail" {
    if case .results(let list) = search.state {
      let detailViewController = segue.destination
                                              as! DetailViewController
      let indexPath = sender as! IndexPath
      let searchResult = list[indexPath.row]
      detailViewController.searchResult = searchResult
    }
  }
}
```

Here you only care about the `.results` case, so writing an entire `switch` statement is a bit much. For situations like this, you can use the special `if case` statement to look at a single case.

There is one more change to make, in **LandscapeViewController.swift**.

➤ Change the `if firstTime` section in `viewWillLayoutSubviews()` to:

```
if firstTime {
  firstTime = false

  switch search.state {
  case .notSearchedYet:
    break
  case .loading:
    break
  case .noResults:
    break
  case .results(let list):
    tileButtons(list)
  }
}
```

This uses the same pattern as before. If the state is `.results`, it binds the array of `SearchResult` objects to the temporary constant `list` and passes it along to

`tileButtons()`. Soon you'll add additional code to the other cases. Because these cases are currently empty, they must contain a `break` statement.

➤ Build and run to see if the app still works. (It should!)

I think enums with associated values are one of the most exciting features of Swift. Here you used them to simplify the way the `Search` state is expressed. No doubt you'll find many other great uses for them in your own apps!

➤ This is a good time to commit your changes.

Spin me right round

If you flip to landscape while the search is still taking place, the app really ought to show an animated spinner to let the user know something is happening.

You're already checking in `viewWillLayoutSubviews()` what the state of the active `Search` object is, so that's an easy fix.

➤ In **LandscapeViewController.swift**, in `viewWillLayoutSubviews()` change the `.loading` case in the `switch` statement to:

```
case .loading:
  showSpinner()
```

If the `Search` object is in the `.loading` state, you need to show the activity spinner.

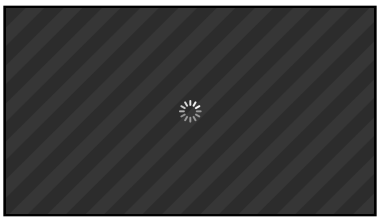
➤ Also add the new `showSpinner()` method:

```
private func showSpinner() {
  let spinner = UIActivityIndicatorView(
                                  activityIndicatorStyle: .whiteLarge)
  spinner.center = CGPoint(x: scrollView.bounds.midX + 0.5,
                           y: scrollView.bounds.midY + 0.5)
  spinner.tag = 1000
  view.addSubview(spinner)
  spinner.startAnimating()
}
```

This programmatically creates a new `UIActivityIndicatorView` object (a big white one this time), puts it in the center of the screen, and starts animating it.

You give the spinner the tag 1000, so you can easily remove it from the screen once the search is done.

➤ Run the app. After starting a search, quickly flip the phone to landscape. You should now see a spinner.



A spinner indicates a search is still taking place

> **Note:** You added `0.5` to the spinner's `center` position. This kind of spinner is 37 points wide and high, which is not an even number. If you were to place the center of this view at the exact center of the screen at (284, 160) then it would extend 18.5 points to either end. The top-left corner of that spinner is at coordinates (265.5, 141.5), making it look all blurry.
>
> It's best to avoid placing objects at fractional coordinates. By adding 0.5 to both the X and Y position, the spinner is placed at (266, 142) and everything looks sharp. Pay attention to this when working with the `center` property and objects that have odd widths or heights.

This is all great, but the spinner doesn't disappear when the actual search results are received. The app never notifies the `LandscapeViewController` of this.

There is a variety of ways you can choose to tell the `LandscapeViewController` that the search results have come in, but let's keep it simple.

➤ In **LandscapeViewController.swift**, add these two new methods:

```
func searchResultsReceived() {
  hideSpinner()

  switch search.state {
  case .notSearchedYet, .loading, .noResults:
    break
  case .results(let list):
    tileButtons(list)
  }
}

private func hideSpinner() {
  view.viewWithTag(1000)?.removeFromSuperview()
}
```

The private `hideSpinner()` method looks for the view with tag 1000 – the activity spinner – and then tells that view to remove itself from the screen.

You could have kept a reference to the spinner in an instance variable but for a simple situation such as this you might as well use a tag. Because no one else has any strong references to the `UIActivityIndicatorView`, this instance will be deallocated. Note that you have to use optional chaining because `viewWithTag()` can potentially return `nil`.

The `searchResultsReceived()` method should be called from somewhere, of course, and that somewhere is the `SearchViewController`.

➤ In **SearchViewController.swift**'s `performSearch()` method, add the following line into the closure (below `self.tableView.reloadData()`):

```
self.landscapeViewController?.searchResultsReceived()
```

The sequence of events here is quite interesting. When the search begins there is no `LandscapeViewController` object yet because the only way to start a search is from portrait mode.

But by the time the closure is invoked, the device may have rotated and if that happened `self.landscapeViewController` will contain a valid reference.

Upon rotation you also gave the new `LandscapeViewController` a reference to the active `Search` object. Now you just have to tell it that search results are available so it can create the buttons and fill them up with images.

Of course, if you're still in portrait mode by the time the search completes then `self.landscapeViewController` is `nil` and the call to `searchResultsReceived()` will simply be ignored due to the optional chaining. (You could have used `if let` here to unwrap the value of `self.landscapeViewController`, but optional chaining has the same effect and is shorter to write.)

➤ Try it out. That works pretty well, eh?

Exercise. Verify that network errors are also handled correctly when the app is in landscape orientation. Find a way to create – or fake! – a network error and see what happens in landscape mode. Hint: the `sleep(5)` function will put your app to sleep for 5 seconds. Put that in the completion handler to give yourself some time to flip the device around. ■

Speaking of spinners, you've probably noticed that your iPhone's status bar shows a small, animated spinner when network activity is taking place. This isn't automatic – the app needs to explicitly turn this animation on or off. Fortunately, it's only a single line of code.

➤ In **Search.swift**, first `import UIKit` (all the way at the top of the file):

```
import UIKit
```

➤ Add the following line to `performSearch(for, category, completion)`, just before starting the search:

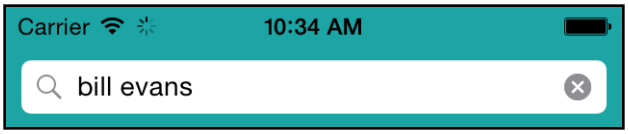
```
func performSearch(for text: String, category: Category,
                   completion: @escaping SearchComplete) {
  if !text.isEmpty {
    dataTask?.cancel()
    UIApplication.shared.isNetworkActivityIndicatorVisible = true
```

```
    . . .
```

This makes the animated spinner visible in the app's status bar. To turn it off again, change the code in `DispatchQueue.main.async` to the following:

```
DispatchQueue.main.async {
  self.state = newState
  UIApplication.shared.isNetworkActivityIndicatorVisible = false
  completion(success)
}
```

➤ Try it out. The app now also shows a spinning animation in the status bar while the search is taking place:



The network activity indicator

Nothing found

You're not done yet. If there are no matches found, you should also tell the user about this if they're in landscape mode.

➤ Inside the `switch` statement in `viewWillLayoutSubviews()`, change the case for `.noResults` to the following:

```
case .noResults:
  showNothingFoundLabel()
```

If there are no search results, you'll call the new `showNothingFoundLabel()` method.

➤ Here is that method:

```
private func showNothingFoundLabel() {
  let label = UILabel(frame: CGRect.zero)
  label.text = "Nothing Found"
  label.textColor = UIColor.white
  label.backgroundColor = UIColor.clear

  label.sizeToFit()

  var rect = label.frame
  rect.size.width = ceil(rect.size.width/2) * 2    // make even
  rect.size.height = ceil(rect.size.height/2) * 2  // make even
  label.frame = rect

  label.center = CGPoint(x: scrollView.bounds.midX,
                         y: scrollView.bounds.midY)
  view.addSubview(label)
}
```

Here you first create a `UILabel` object and give it text and a color. To make the label see-through the `backgroundColor` property is set to `UIColor.clear`.

The call to `sizeToFit()` tells the label to resize itself to the optimal size. You could have given the label a frame that was big enough to begin with, but I find this just as easy. (It also helps when you're translating the app to a different language, in which case you may not know beforehand how large the label needs to be.)

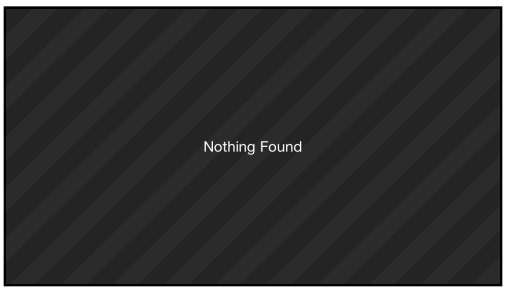
The only trouble is that you want to center the label in the view and as you saw before that gets tricky when the width or height are odd (something you don't necessarily know in advance). So here you use a little trick to always force the dimensions of the label to be even numbers:

```
width = ceil(width/2) * 2
```

If you divide a number such as 11 by 2 you get 5.5. The `ceil()` function rounds up 5.5 to make 6, and then you multiply by 2 to get a final value of 12. This formula always gives you the next even number if the original is odd. (You only need to do this because these values have type `CGFloat`. If they were integers, you wouldn't have to worry about fractional parts.)

> **Note:** Because you're not using a hardcoded number such as 480 or 568 but `scrollView.bounds` to determine the width of the screen, the code to center the label works correctly on all iPhone models.

➤ Run the app and search for something ridiculous (**ewdasuq3sadf843** will do). When the search is done, flip to landscape.



Yup, nothing found here either

It doesn't work properly yet when you flip to landscape while the search is taking place. Of course you also need to put some logic in `searchResultsReceived()`.

➤ Change the `switch` statement in that method to:

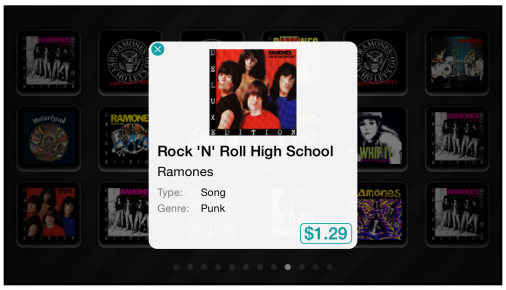
```
switch search.state {
case .notSearchedYet, .loading:
  break
```

```
  case .noResults:
    showNothingFoundLabel()
  case .results(let list):
    tileButtons(list)
  }
```

Now you should have all your bases covered.

The Detail pop-up

These landscape search results are not buttons for nothing. The app should show the Detail pop-up when you tap them, like this:



The pop-up in landscape mode

This is fairly easy to achieve. When adding the buttons you can give them a **target-action**, i.e. a method to call when the Touch Up Inside event is received. Just like in Interface Builder, except now you hook up the event to the action method programmatically.

➤ Add the following two lines to the button creation code in `tileButtons()`:

```
  button.tag = 2000 + index
  button.addTarget(self, action: #selector(buttonPressed),
                   for: .touchUpInside)
```

First you give the button a tag, so you know to which index in the `.results` array this button corresponds. That's needed in order to pass the correct `SearchResult` object to the Detail pop-up.

Tip: You added 2000 to the index because tag 0 is used on all views by default so asking for a view with tag 0 might actually return a view that you didn't expect. To avoid this kind of confusion, you simply start counting from 2000. You also tell the button it should call a method named `buttonPressed()` when it gets tapped.

➤ Add this new `buttonPressed()` method:

```
  func buttonPressed(_ sender: UIButton) {
    performSegue(withIdentifier: "ShowDetail", sender: sender)
  }
```

Even though this is an action method you didn't declare it as `@IBAction`. That is only necessary when you want to connect the method to something in Interface Builder. Here you made the connection programmatically, so you can skip the `@IBAction` annotation.

Pressing the button triggers a segue, which means you need a *prepare-for-segue* to do all the work:

➤ Add the `prepare(for, sender)` method:

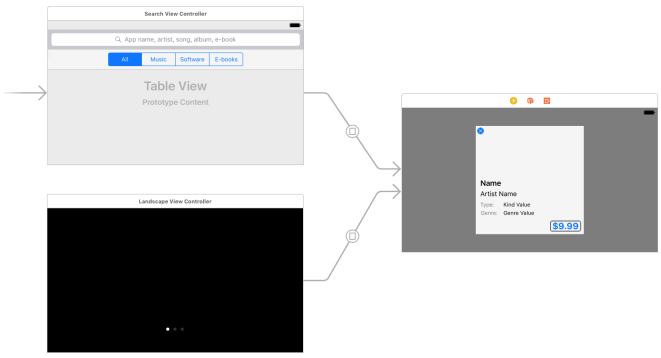
```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
  if segue.identifier == "ShowDetail" {
    if case .results(let list) = search.state {
      let detailViewController = segue.destination
                                              as! DetailViewController
      let searchResult = list[(sender as! UIButton).tag - 2000]
      detailViewController.searchResult = searchResult
    }
  }
}
```

This is almost word-for-word identical to `prepare(for:sender:)` from `SearchViewController`, except now you don't get the index of the `SearchResult` object from an index-path but from the button's tag (minus 2000).

Of course, none of this will work unless you actually make a segue in the storyboard first.

➤ Go to the Landscape View Controller in the storyboard and Ctrl-drag to the Detail View Controller. Make it a **Present Modally** segue with identifier **ShowDetail**.

The storyboard looks like this now:



The storyboard after connecting the Landscape view to the Detail pop-up

➤ Run the app and check it out.

Cool. But what happens when you rotate back to portrait with a Detail pop-up showing? Unfortunately, it sticks around. You still need to tell the Detail screen to close.

➤ In **SearchViewController.swift**, in `hideLandscape(with)`, add the following lines to the `animate(alongsideTransition)` animation closure:

```
if self.presentedViewController != nil {
  self.dismiss(animated: true, completion: nil)
}
```

In the debug pane output you should see that the `DetailViewController` is properly deallocated when you rotate back to portrait.

➤ If you're happy with the way it works, then let's commit it. If you also made a branch, then merge it back into the master branch.

You can find the project files for the app under **08 - Refactored Search** in the tutorial's Source Code folder.

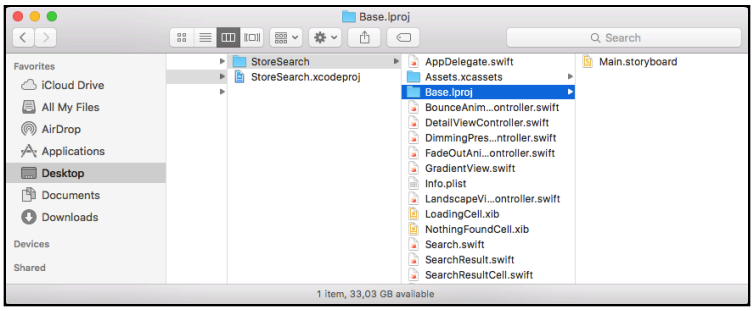
Internationalization

So far the apps you've made in this tutorial series have all been in English. No doubt the United States is the single biggest market for apps, followed closely by Asia. But even if you add up all the smaller countries where English isn't the primary language, you still end up with quite a sizable market that you might be missing out on.

Fortunately, iOS makes it very easy to add support for other languages to your apps, a process known as **internationalization**. This is often abbreviated as "i18n" because that's a lot shorter to write; the 18 stands for the number of letters between the i and the n. You'll also often hear the word **localization**, which basically means the same thing.

In this section you'll add support for Dutch, which is my native language. You'll also make the web service query return results that are optimized for the user's regional settings.

The structure of your source code folder probably looks something like this:



The files in the source code folder

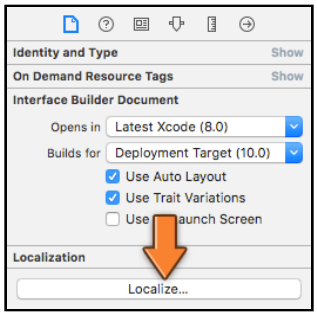
There is a subfolder named **Base.lproj** that contains one file, **Main.storyboard**. The Base.lproj folder is for files that can be localized. So far that's only the storyboard but you'll add more files to this folder soon.

When you add support for another language, a new **XX.lproj** folder is created with XX being the two-letter code for that new language (**en** for English, **nl** for Dutch).

Let's begin by localizing a simple file, the **NothingFoundCell.xib**. Often nib files contain text that needs to be translated. You can simply make a new copy of the existing nib file for a specific language and put it in the right .lproj folder. When the iPhone is using that language, it will automatically load the translated nib.

➤ Select **NothingFoundCell.xib** in the Project navigator. Switch to the **File inspector** pane (on the right of the Xcode window).

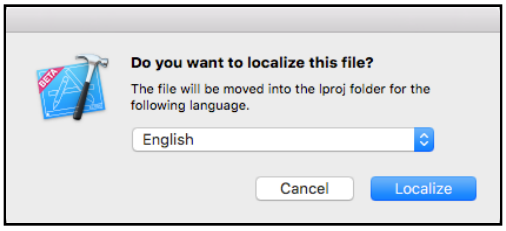
Because the NothingFoundCell.xib file isn't in any XX.lproj folders, it does not have any localizations yet.



The NothingFoundCell has no localizations

➤ Click the **Localize...** button in the Localization section.

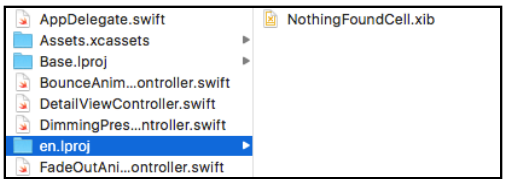
Xcode asks for confirmation because this involves moving the file to a new folder.



Xcode asks whether it's OK to move the file

➤ Choose **English** (not Base) and click **Localize** to continue.

Look in Finder and you will see there is a new **en.lproj** folder (for English) and NothingFoundCell.xib has moved into that folder.



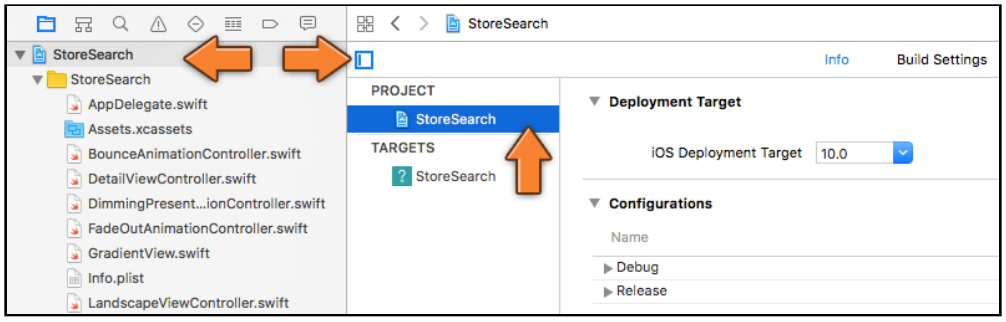
Xcode moved NothingFoundCell.xib to the en.lproj folder

The **File inspector** for **NothingFoundCell.xib** now lists English as one of the localizations.

The Localization section now contains an entry for English

To add a new language you have to switch to the **Project Settings** screen.

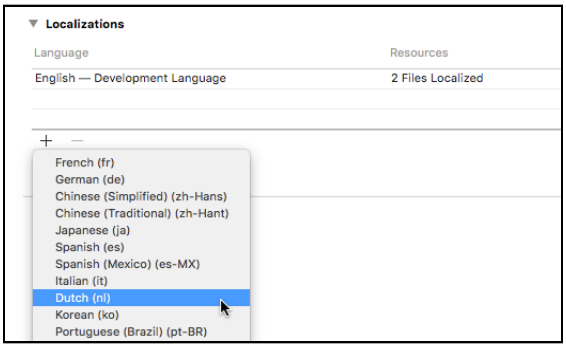
➤ Click on **StoreSearch** at the top of the Project navigator to open the settings page.



The Project Settings

➤ From the sidebar, choose **StoreSearch** under **PROJECT** (not under TARGETS). If the sidebar isn't visible click the small icon at the top to open it.

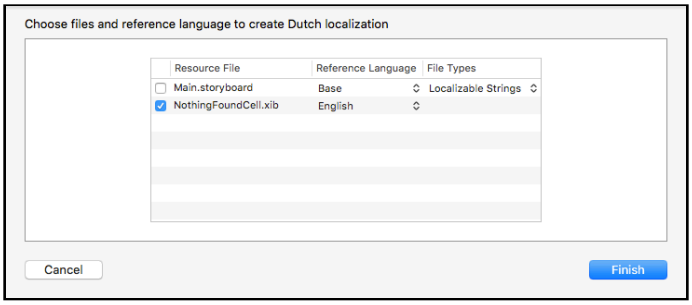
➤ In the **Info** tab, under the **Localizations** section press the **+** button:



Adding a new language

➤ From the pop-up menu choose **Dutch (nl)**.

Xcode now asks which resources you want to localize. Uncheck everything except for **NothingFoundCell.xib** and click **Finish**.



Choosing the files to localize

If you look in Finder again you'll notice that a new subfolder has been added, **nl.lproj**, and that it contains another copy of NothingFoundCell.xib.

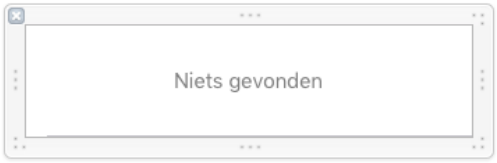
That means there are now two nib files for `NothingFoundCell`. You can also see this in the Project navigator:

NothingFoundCell.xib has two localizations

Let's edit the new Dutch version of this nib.

➤ Click on **NothingFoundCell.xib (Dutch)** to open it in Interface Builder.

➤ Change the label text to **Niets gevonden** and center the label again in the view (if necessary, use the Update Frames button or the Resolve Auto Layout Issues menu).



That's how you say it in Dutch

It is perfectly all right to resize or move around items in a translated nib. You could make the whole nib look completely different if you wanted to (but that's probably a bad idea). Some languages, such as German, have very long words and in those cases you may have to tweak label sizes and fonts to get everything to fit.

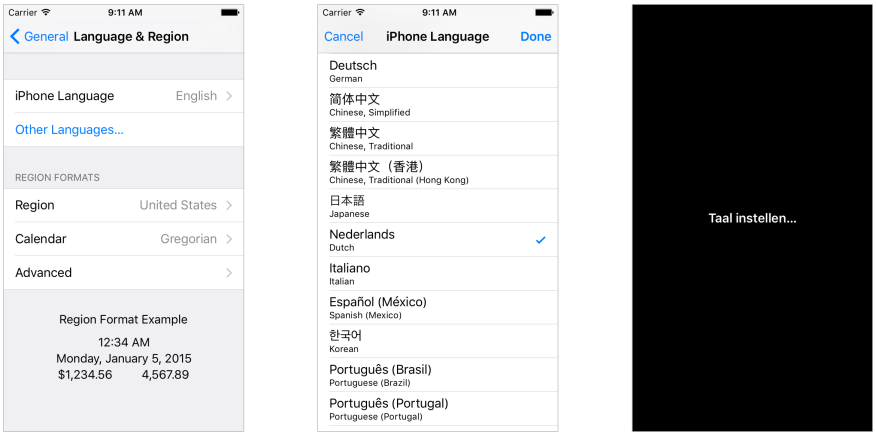
If you run the app now, nothing will have changed. You have to switch the Simulator to use the Dutch language first. However, before you do that you really should remove the app from the simulator, clean the project, and do a fresh build.

The reason for this is that the nibs were previously not localized. If you were to switch the simulator's language now, the app would still keep using the old, non-localized versions of the nibs.

> **Note:** For this reason it's a good idea to already put all your nib files and storyboards in the **en.lproj** folder when you create them (or in **Base.lproj**, which we'll discuss shortly). Even if you don't intend to internationalize your app any time soon, you don't want your users to run into the same problem later on. It's not nice to ask your users to uninstall the app – and lose their data – in order to be able to switch languages.

➤ Remove the app from the Simulator. Do a clean (**Product → Clean** or **Shift-⌘-K**) and re-build the app.

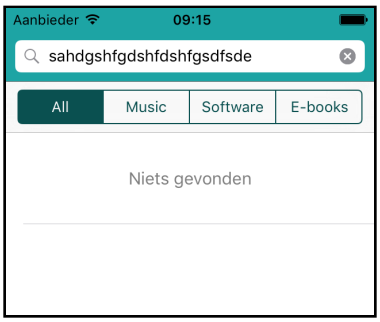
➤ Open the **Settings** app in the Simulator and go to **General → Language & Region → iPhone Language**. From the list pick **Nederlands (Dutch)**.



Switching languages in the Simulator

The Simulator will take a moment to switch between languages. This terminates the app if it was still running.

➤ Search for some nonsense text and the app will now respond in Dutch.



I'd be surprised if that did turn up a match

Pretty cool. Just by placing some files in the **en.lproj** and **nl.lproj** folders, you have internationalized the app. You're going to keep the Simulator in Dutch for a while because the other nibs need translating too.

> **Note:** If the app crashes for you at this point, then the following might help. Quit Xcode. Reset the Simulator and then quit it. In Finder go to your **Library** folder, **Developer**, **Xcode** and throw away the entire **DerivedData** folder. Empty your trashcan. Then open the StoreSearch project again and give it another try. (Don't forget to switch the Simulator back to **Nederlands**.)

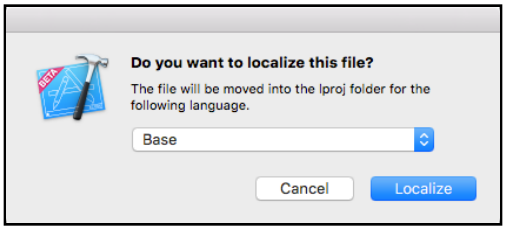
To localize the other nibs you could repeat the process and add copies of their xib files to the **nl.lproj** folder. That isn't too bad for this app but if you have an app with really complicated screens then having multiple copies of the same nib can become a maintenance nightmare.

Whenever you need to change something to that screen you need to update all of those nibs. There's a risk that you forget one nib and they go out-of-sync. That's just asking for bugs – in languages that you probably don't speak!

To prevent this from happening you can use **base internationalization**. With this feature enabled you don't copy the entire nib, but only the text strings. This is what the **Base.lproj** folder is for.

Let's translate the other nibs.

➤ Open **LoadingCell.xib** in Interface Builder. In the **File inspector** press the **Localize...** button. This time use **Base** as the language:



Choosing the Base localization as the destination

Verify with Finder that LoadingCell.xib got moved into the **Base.lproj** folder.

➤ The Localization section in the **File inspector** for **LoadingCell.xib** now contains three options: Base (with a checkmark), English, and Dutch. Put a checkmark in front of **Dutch**:

Adding a Dutch localization

In Finder you can see that **nl.proj** doesn't get a copy of the nib, but a new type of file: **LoadingCell.strings**.

➤ Click the arrow in front of **LoadingCell.xib** to expand it in the Project navigator and open the **LoadingCell.strings (Dutch)** file.

You should see something like the following:

The Dutch localization is a strings file

There is still only one nib, the one from the Base localization. The Dutch translation consists of a "strings" file with just the texts from the labels, buttons, and other controls.

The contents of this particular strings file are:

```
/* Class = "UILabel"; text = "Loading..."; ObjectID = "9y5-pI-mmH"; */
"9y5-pI-mmH.text" = "Loading...";
```

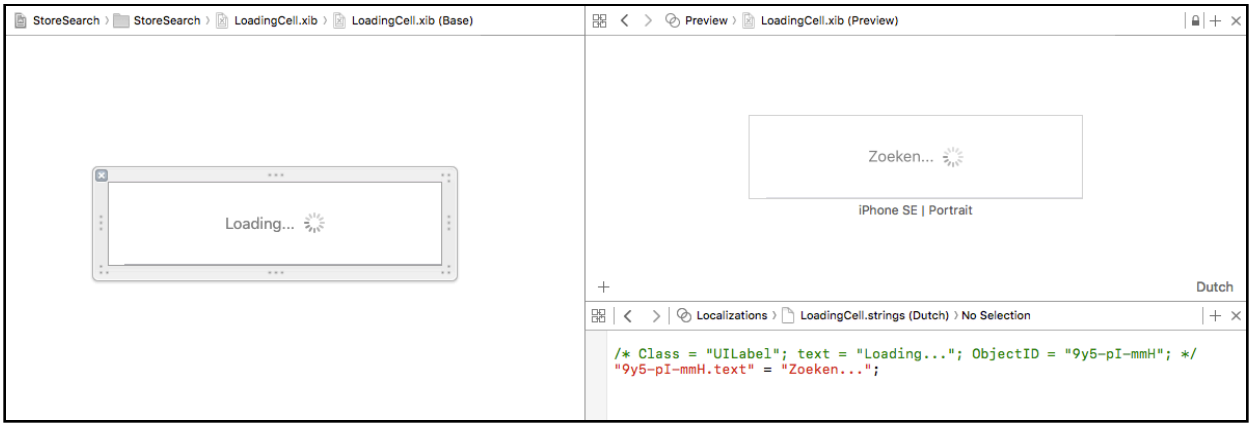
The green bit is a comment, just like in Swift. The second line says that the **text** property of the object with ID "9y5-pI-mmH" contains the text **Loading...**

The ID is an internal identifier that Xcode uses to keep track of the objects in your nibs; your own nib probably has a different code than mine. You can see this ID in the Identity inspector for the label.

➤ Change the text **Loading...** into **Zoeken...**

Tip: You can use the Assistant editor in Interface Builder to get a preview of your localized nib. Go to **LoadingCell.xib (Base)** and open the Assistant editor.

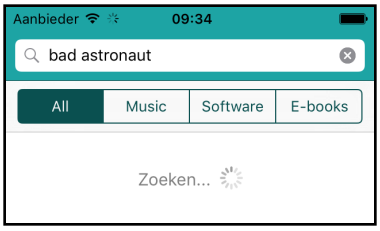
From the Jump bar at the top, choose **Preview**. In the bottom-right corner it says English. Click this to switch to a Dutch preview.



The Assistant editor shows a preview of the translation

If you open a second assistant pane (with the **+**) and set that to **Localizations**, you can edit the translations and see what they look like at the same time. Very handy!

➤ Do a **Product → Clean** and run the app again.



The localized loading text

> **Note:** If you don't see the "Zoeken…" text then do the same dance again: quit Xcode, throw away the DerivedData folder, reset the Simulator.

➤ Repeat the steps to add a Dutch localization for **Main.storyboard**. It already has a Base localization so you simply have to put a check in front of **Dutch** in the File inspector.

For the Search View Controller screen two things need to change: the placeholder text in the Search Bar and the labels on the Segmented Control.

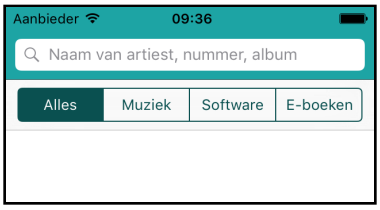
➤ In **Main.strings (Dutch)** change the placeholder text to **Naam van artiest, nummer, album**.

```
"68e-CH-NSs.placeholder" = "Naam van artiest, nummer, album";
```

The segment labels will become: **Alles**, **Muziek**, **Software**, and **E-boeken**.

```
"Sjk-fv-Pca.segmentTitles[0]" = "Alles";
"Sjk-fv-Pca.segmentTitles[1]" = "Muziek";
"Sjk-fv-Pca.segmentTitles[2]" = "Software";
"Sjk-fv-Pca.segmentTitles[3]" = "E-boeken";
```

(Of course your object IDs will be different.)



The localized SearchViewController

➤ For the Detail pop-up you only need to change the **Type:** label to say **Soort:**

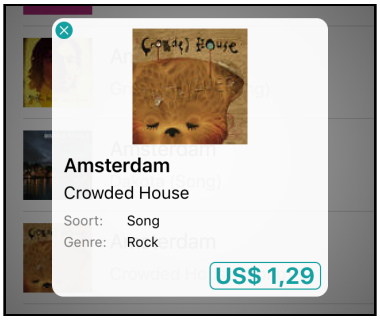
```
"DCQ-US-EVg.text" = "Soort:";
```

You don't need to change these:

```
"ZYp-Zw-Fg6.text" = "Genre:";
"yz2-Gh-kzt.text" = "Kind Value";
"Ph9-wm-1LS.text" = "Artist Name";
"JVj-dj-Iz8.text" = "Name";
"7sM-UJ-kWH.text" = "Genre Value";
"xOH-GC-bHs.normalTitle" = "$9.99";
```

These labels can remain the same because you will replace them with values from the `SearchResult` object anyway. ("Genre" is the same in both languages.)

> **Note:** If you wanted to, you could even remove the texts that don't need localization from the strings file. If a localized version for a specific resource is missing for the user's language, iOS will fall back to the one from the Base localization.



The pop-up in Dutch

Thanks to Auto Layout, the labels automatically resize to fit the translated text. A common issue with localization is that English words tend to be shorter than words in other languages so you have to make sure your labels are big enough. With Auto Layout that is a piece of cake.

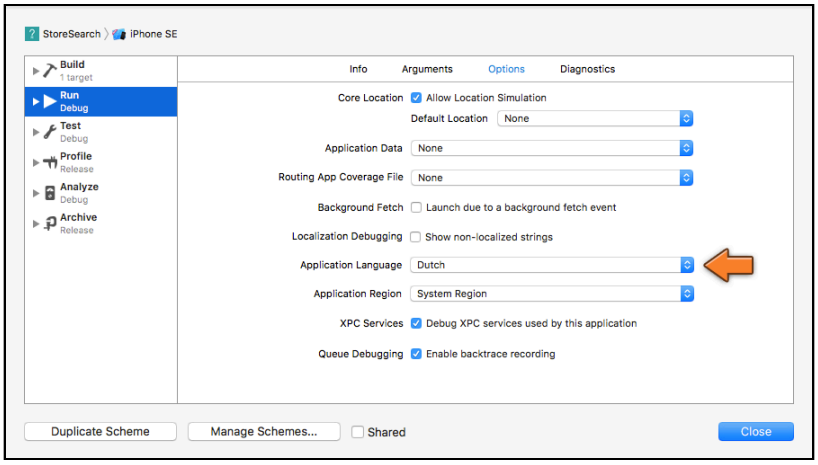
The Landscape View Controller doesn't have any text to translate.

➤ There is no need to give **SearchResultCell.xib** a Dutch localization (there is no on-screen text in the nib itself) but do give it a Base localization. This prepares the app for the future, should you need to localize this nib at some point.

When you're done there should be no more **xib** files outside the **.lproj** folders.

That's it for the nibs and the storyboard. Not so bad, was it? I'd say all these changes are commit-worthy.

Tip: You can also test localizations by changing the settings for the active scheme. Click on **StoreSearch** in the Xcode toolbar (next to the Simulator name) and choose **Edit Scheme**.



In the **Options** tab you can change the **Application Language** and **Region** settings. That's a bit quicker than restarting the Simulator.

Localizing on-screen texts

Even though the nibs and storyboard have been translated, not all of the text is. For example, in the image above the text from the `kind` property is still "Song".

While in this case you could get away with it – probably everyone in the world knows what the word "Song" means – not all of the texts from the `kindForDisplay()` method will be understood by non-English speaking users.

To localize texts that are not in a nib or storyboard, you have to use another approach.

➤ In **SearchResult.swift**, make sure the Foundation framework is imported:

```
import Foundation
```

➤ Then replace the `kindForDisplay()` method with:

```
func kindForDisplay() -> String {
  switch kind {
  case "album":
    return NSLocalizedString("Album", comment: "Localized kind: Album")
  case "audiobook":
    return NSLocalizedString("Audio Book",
                             comment: "Localized kind: Audio Book")
  case "book":
    return NSLocalizedString("Book", comment: "Localized kind: Book")
  case "ebook":
    return NSLocalizedString("E-Book",
                             comment: "Localized kind: E-Book")
  case "feature-movie":
    return NSLocalizedString("Movie",
                             comment: "Localized kind: Feature Movie")
  case "music-video":
    return NSLocalizedString("Music Video",
                             comment: "Localized kind: Music Video")
  case "podcast":
    return NSLocalizedString("Podcast",
                             comment: "Localized kind: Podcast")
  case "software":
    return NSLocalizedString("App", comment: "Localized kind: Software")
  case "song":
    return NSLocalizedString("Song", comment: "Localized kind: Song")
  case "tv-episode":
    return NSLocalizedString("TV Episode",
                             comment: "Localized kind: TV Episode")
  default:
    return kind
  }
}
```

Tip: Rather than typing in the above you can use Xcode's powerful Regular Expression Replace feature to make those changes in just a few seconds.

Go to the **Search inspector** and change its mode from Find to **Replace > Regular Expression**.

In the search box type: **return "(.+)"**

In the replacement box type:

return NSLocalizedString("$1", comment: "Localized kind: $1")

Press **enter** to search. This looks for any lines that match the pattern *return "something"*. Whatever that *something* is will be put in the $1 placeholder of the replacement text.

> Click **Preview**. Make sure only **SearchResult.swift** is selected – you don't want to make this change in any of the other files! Click **Replace** to finish.
>
> Thanks to Scott Gardner for the tip!

The structure of `kindForDisplay()` is still the same as before, but instead of doing,

```
return "Album"
```

it now does:

```
return NSLocalizedString("Album", comment: "Localized kind: Album")
```

Slightly more complicated but also a lot more flexible.

`NSLocalizedString()` takes two parameters: the text to return, `"Album"`, and a comment, `"Localized kind: Album"`.

Here is the cool thing: if your app includes a file named **Localizable.strings** for the user's language, then `NSLocalizedString()` will look up the text ("Album") and returns the translation as specified in Localizable.strings.

If no translation for that text is present, or there is no Localizable.strings file, then `NSLocalizedString()` simply returns the text as-is.

➤ Run the app again. The "Type:" field in the pop-up (or "Soort:" in Dutch) should still show the same kind of texts as before because you haven't translated anything yet.

To create the **Localizable.strings** file, you will use a command line tool named **genstrings**. This requires a trip to the Terminal.

➤ Open a Terminal, `cd` to the folder that contains the StoreSearch project. You want to go into the folder that contains the actual source files. On my system that is:

```
cd ~/Desktop/StoreSearch/StoreSearch
```

Then type the following command:

```
genstrings *.swift -o en.lproj
```

This looks at all your source files (*.**swift**) and writes a new file called **Localizable.strings** in the **en.lproj** folder.

➤ Add this **Localizable.strings** file from the **en.lproj** folder to the project in Xcode. (To be safe, disable **Copy items if needed**. You want to add the file from en.lproj, not make a copy.)

If you open the Localizable.strings file, this is what it currently contains:

```
/* Localized kind: Album */
"Album" = "Album";

/* Localized kind: Software */
"App" = "App";

/* Localized kind: Audio Book */
"Audio Book" = "Audio Book";

/* Localized kind: Book */
"Book" = "Book";

/* Localized kind: E-Book */
"E-Book" = "E-Book";

/* Localized kind: Feature Movie */
"Movie" = "Movie";

/* Localized kind: Music Video */
"Music Video" = "Music Video";

/* Localized kind: Podcast */
"Podcast" = "Podcast";

/* Localized kind: Song */
"Song" = "Song";

/* Localized kind: TV Episode */
"TV Episode" = "TV Episode";
```

The things between the `/*` and `*/` symbols are the comments you specified as the second parameter of `NSLocalizedString()`. They give the translator some context about where the string is supposed to be used in the app.

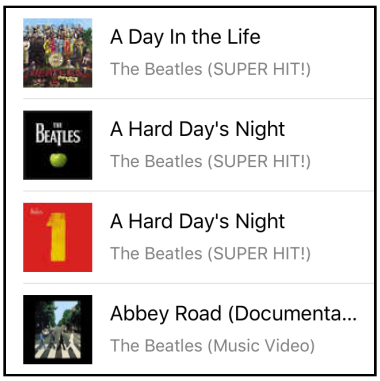
Tip: It's a good idea to make these comments as detailed as you can. In the words of fellow tutorial author Scott Gardner:

"The comment to the translator should be as detailed as necessary to not only state the words to be transcribed, but also the perspective, intention, gender frame of reference, etc. Many languages have different words based on these considerations. I translated an app into Chinese Simplified once and it took multiple passes to get it right because my original comments were not detailed enough."

➤ Change the "Song" line to:

```
"Song" = "SUPER HIT!";
```

➤ Now run the app again and search for music. For any search result that is a song, it will now say "SUPER HIT!" instead.



Where it used to say Song it now says SUPER HIT!

Of course, changing the texts in the English localization doesn't make much sense, so put Song back to what it was and then we'll do it properly.

➤ In the **File inspector**, add a Dutch localization for this file. This creates a copy of Localizable.strings in the **nl.lproj** folder.

➤ Change the translations in the Dutch version of **Localizable.strings** to:

```
"Album" = "Album";
"App" = "App";
"Audio Book" = "Audioboek";
"Book" = "Boek";
"E-Book" = "E-Boek";
"Movie" = "Film";
"Music Video" = "Videoclip";
"Podcast" = "Podcast";
"Song" = "Liedje";
"TV Episode" = "TV serie";
```

If you run the app again, the product types will all be in Dutch. Nice!

Always use NSLocalizedString() from the beginning

There are a bunch of other strings in the app that need translation as well. You can search for anything that begins with " but it would have been a lot easier if we had used `NSLocalizedString()` from the start. Then all you would've had to do was run the **genstrings** tool and you'd get all the strings.

Now you have to comb through the source code and add `NSLocalizedString()` everywhere there is text that will be shown to the user. (Mea culpa!)

You should really get into the habit of always using `NSLocalizedString()` for strings that you want to display to the user, even if you don't care about internationalization right away.

Adding support for other languages is a great way for your apps to become more popular, and going back through your code to add `NSLocalizedString()` is not much fun. It's better to do it right from the start!

Here are the other strings I found that need to be `NSLocalizedString`-ified:

```
// DetailViewController, updateUI()
artistNameLabel.text = "Unknown"
priceText = "Free"

// SearchResultCell, configure(for)
artistNameLabel.text = "Unknown"

// LandscapeViewController, showNothingFoundLabel()
label.text = "Nothing Found"

// SearchViewController, showNetworkError()
title: "Whoops...",
message: "There was an error reading from the iTunes Store.
          Please try again.",
title: "OK"
```

➤ Add `NSLocalizedString()` around these texts. Don't forget to use descriptive comments!

For example, when instantiating the `UIAlertController` in `showNetworkError()`, you could write:

```
let alert = UIAlertController(
  title: NSLocalizedString("Whoops...", comment: "Error alert: title"),
  message: NSLocalizedString(
  "There was an error reading from the iTunes Store. Please try again.",
                                        comment: "Error alert: message"),
  preferredStyle: .alert)
```

Note: You don't need to use `NSLocalizedString()` with your `print()`'s. Debug output is really intended only for you, the developer, so it's best if it is in English (or your native language).

➤ Run the **genstrings** tool again. Give it the same arguments as before. It will put a clean file with all the new strings in the **en.lproj** folder.

Unfortunately, there really isn't a good way to make genstrings merge new strings into existing translations. It will overwrite your entire file and throw away any changes that you made. There is a way to make the tool append its output to an existing file but then you end up with a lot of duplicate strings.

> **Tip:** Always regenerate only the file in en.lproj and then copy over the missing strings to your other Localizable.strings files. You can use a tool such as FileMerge or Kaleidoscope to compare the two to see where the new strings are. There are also several third-party tools on the Mac App Store that are a bit friendlier to use than genstrings.

➤ Add these new translations to the Dutch **Localizable.strings**:

```
"Nothing Found" = "Niets gevonden";

"There was an error reading from the iTunes Store. Please try again." =
"Er ging iets fout bij het communiceren met de iTunes winkel. Probeer het
nog eens.";

"Unknown" = "Onbekend";

"Whoops..." = "Foutje...";
```

It may seem a little odd that such as long string as "There was an error reading from the iTunes Store. Please try again." would be used as the lookup key for a translated string, but there really isn't anything wrong with it.

(By the way, the semicolons at the end of each line are not optional. If you forget a semicolon, the Localizable.strings file cannot be compiled and the build will fail.)

Some people write code like this:

```
let s = NSLocalizedString("ERROR_MESSAGE23",
                          comment: "Error message on screen X")
```

The Localizable.strings file would then look like:

```
/* Error message on screen X */
"ERROR_MESSAGE23" = "Does not compute!";
```

This works but I find it harder to read. It requires that you always have an English Localizable.strings as well. In any case, you will see both styles used in practice.

Note also that the text `"Unknown"` occurred only once in Localizable.strings even though it shows up in two different places in the source code. Each piece of text only needs to be translated once.

If your app builds strings dynamically, then you can also localize such texts. For example in **SearchResultCell.swift**, `configure(for)` you do:

```
artistNameLabel.text = String(format: "%@ (%@)",
                searchResult.artistName, searchResult.kindForDisplay())
```

➤ Internationalize this as follows:

```
artistNameLabel.text = String(format:
  NSLocalizedString("%@ (%@)", comment: "Format for artist name label"),
                searchResult.artistName, searchResult.kindForDisplay())
```

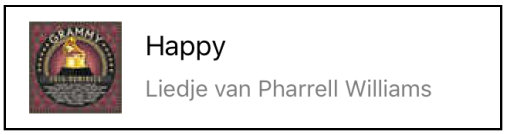
After running **genstrings** again, this shows up in Localizable.strings as:

```
/* Format for artist name label */
"%@ (%@)" = "%1$@ (%2$@)";
```

If you wanted to, you could change the order of these parameters in the translated file. For example:

```
"%@ (%@)" = "%2$@ van %1$@";
```

It will turn the artist name label into something like this:



The "kind" now comes first, the artist name last

In this circumstance I would advocate the use of a special key rather than the literal string to find the translation. It's thinkable that your app will employ the format string `"%@ (%@)"` in some other place and you may want to translate that completely differently there.

I'd call it something like `"ARTIST_NAME_LABEL_FORMAT"` instead (this goes in the Dutch Localizable.strings):

```
/* Format for artist name label */
"ARTIST_NAME_LABEL_FORMAT" = "%2$@ van %1$@";
```

You also need to add this key to the English version of Localizable.strings:

```
/* Format for artist name label */
"ARTIST_NAME_LABEL_FORMAT" = "%1$@ (%2$@)";
```

Don't forget to change the code as well:

```
artistNameLabel.text = String(format:
            NSLocalizedString("ARTIST_NAME_LABEL_FORMAT",
                      comment: "Format for artist name label"),
                searchResult.artistName, searchResult.kindForDisplay())
```

There is one more thing I'd like to improve. Remember how in **SearchResult.swift** the `kindForDisplay()` method is this enormous `switch` statement? That's "smelly" to me. The problem is that any new products require you to add need another case to the switch.

For situations like these it's better to use a *data-driven* approach. Here that means you place the product types and their human-readable names in a data structure, a dictionary, rather than a code structure.

➤ Add the following dictionary to **SearchResult.swift**, above the `class` (you may want to copy-paste this from `kindForDisplay()` as it's almost identical):

```
private let displayNamesForKind = [
  "album": NSLocalizedString("Album", comment: "Localized kind: Album"),
  "audiobook": NSLocalizedString("Audio Book",
                                 comment: "Localized kind: Audio Book"),
  "book": NSLocalizedString("Book", comment: "Localized kind: Book"),
  "ebook": NSLocalizedString("E-Book",
                             comment: "Localized kind: E-Book"),
  "feature-movie": NSLocalizedString("Movie",
                              comment: "Localized kind: Feature Movie"),
  "music-video": NSLocalizedString("Music Video",
                               comment: "Localized kind: Music Video"),
  "podcast": NSLocalizedString("Podcast",
                              comment: "Localized kind: Podcast"),
  "software": NSLocalizedString("App",
                               comment: "Localized kind: Software"),
  "song": NSLocalizedString("Song",
                            comment: "Localized kind: Song"),
  "tv-episode": NSLocalizedString("TV Episode",
                                  comment: "Localized kind: TV Episode"),
]
```

Now the code for `kindForDisplay()` becomes really short:

```
func kindForDisplay() -> String {
  return displayNamesForKind[kind] ?? kind
}
```

It's nothing more than a simply dictionary lookup.

The `??` is the **nil coalescing** operator. Remember that dictionary lookups always return an optional, just in case the key you're looking for – `kind` in this case – does not exist in the dictionary. That could happen if the iTunes web service added new product types.

If the dictionary gives you `nil`, the `??` operator simply returns the original value of kind. It's equivalent to writing,

```
if let name = displayNamesForKind[kind] {
  return name
} else {
  return kind
}
```

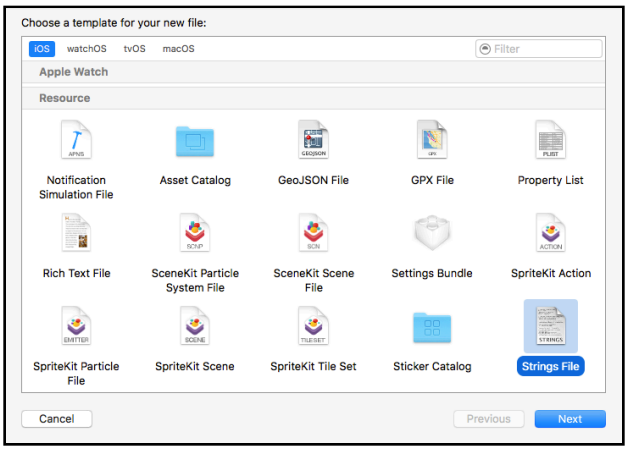
but shorter!

InfoPlist.strings

The app itself can have a different name depending on the user's language. The name that is displayed on the iPhone's home screen comes from the **Bundle name** setting in **Info.plist** or if present, the **Bundle display name** setting.

To localize the texts from Info.plist you need a file named **InfoPlist.strings**.

➤ Add a new file to the project. In the template chooser scroll down to the **Resource** group and choose **Strings File**. Name it **InfoPlist.strings** (the capitalization matters!).



Adding a new Strings file to the project

➤ Open **InfoPlist.strings** and press the **Localize...** button from the File inspector. Choose the **Base** localization.

➤ Also add a **Dutch** localization for this file.

➤ Open the Dutch version and add the following line:

```
CFBundleDisplayName = "StoreZoeker";
```

The key for the "Bundle display name" setting is `CFBundleDisplayName`.

(Dutch readers, sorry for the silly name. This is the best I could come up with. Feel free to substitute your own.)

➤ Run the app and close it so you can see its icon. The Simulator's stringboard should now show the translated app name:

Even the app's name is localized!

If you switch the Simulator back to English, the app name is StoreSearch again (and of course, all the other text is back in English as well).

Regional settings

I don't know if you noticed in some of the earlier screenshots, but even though you switched the language to Dutch, the prices of the products still show up in US dollars instead of Euros. That has two reasons:

1. The language settings are independent of the regional settings. How currencies and numbers are displayed depends on the region settings, not the language.
2. The app does not specify anything about country or language when it sends the requests to the iTunes store, so the web service always returns prices in US dollars.

First you'll fix the app so that it sends information about the user's language and regional settings to the iTunes store. The method that you are going to change is **Search.swift**'s `iTunesURL(searchText, category)` because that's where you construct the parameters that get sent to the web service.

➤ Change the `iTunesURL(searchText, category)` method to the following:

```
private func iTunesURL(searchText: String, category: Category) -> URL {
  let entityName = category.entityName
  let locale = Locale.autoupdatingCurrent
  let language = locale.identifier
  let countryCode = locale.regionCode ?? "en_US"

  let escapedSearchText = searchText.addingPercentEncoding(
                withAllowedCharacters: CharacterSet.urlQueryAllowed)!
  let urlString = String(format: "https://itunes.apple.com/search?
term=%@&limit=200&entity=%@&lang=%@&country=%@", escapedSearchText,
entityName, language, countryCode)

  let url = URL(string: urlString)
  print("URL: \(url!)")
  return url!
}
```

The regional settings are also referred to as the user's **locale** and of course there is an object for it, `Locale`. You get a reference to the `autoupdatingCurrent` locale.

This locale object is called "autoupdating" because it always reflects the current state of the user's locale settings. In other words, if the user changes her regional information while the app is running, the app will automatically use these new settings the next time it does something with that `Locale` object.

From the `locale` object you get the language and the country code. You then put these two values into the URL using the `&lang=` and `&country=` parameters. Because `locale.regionCode` may be `nil`, we use `?? "en_US"` as a failsafe.

The `print()` lets you see what exactly the URL will be.

➤ Run the app and do a search. Xcode should output the following:

```
https://itunes.apple.com/search?
term=bird&limit=200&entity=&lang=en_US&country=US
```

It added "en_US" as the language identifier and just "US" as the country. For products that have descriptions (such as apps) the iTunes web service will return the English version of the description. The prices of all items will have USD as the currency.

> **Note:** It's also possible you got an error message, which happens when the locale identifier returns something nonsensical such as `nl_US`. This is due to the combination of language and region settings on your Mac or the Simulator. If you also change the region (see below), the error should disappear. The iTunes web service does not support all combinations of languages and regions, so an improvement to the app would be to check the value of `language` against a list of allowed languages (left as an exercise for the reader).

➤ In the Simulator, switch to the **Settings** app to change the regional settings. Go to **General → Language & Region → Region**. Select **Netherlands**.

If the Simulator is still in Dutch, then it is under **Algemeen → Taal en Regio → Regio**. Change it to **Nederland**.

➤ Run StoreSearch again and repeat the search.

Xcode now says:

```
https://itunes.apple.com/search?
term=bird&limit=200&entity=&lang=nl_NL&country=NL
```

The language and country have both been changed to NL (for the Netherlands). If you tap on a search result you'll see that the price is now in Euros:

The price according to the user's region settings

Of course, you have to thank `NumberFormatter` for this. It now knows the region settings are from the Netherlands so it uses a comma for the decimal point.

And because the web service now returns `"EUR"` as the currency code, the number formatter puts the Euro symbol in front of the amount. You can get a lot of functionality for free if you know which classes to use!

That's it as far as internationalization goes. It takes only a small bit of effort that definitely pays back. (You can put the Simulator back to English now.)

➤ It's time to commit because you're going to make some big changes in the next section.

If you've also been tagging the code, you can call this v0.9, as you're rapidly approaching the 1.0 version that is ready for release.

The project files for the app up to this point are under **09 - Internationalization** in the tutorial's Source Code folder.

The iPad

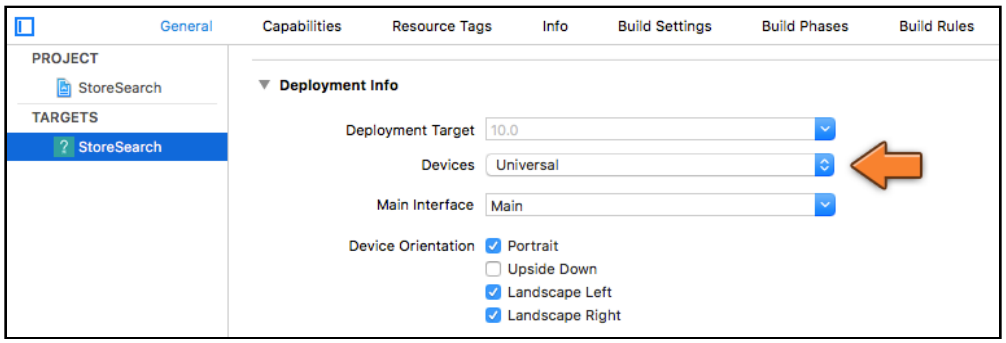
Even though the apps you've written so far are only for the iPhone, everything you have learned also applies to writing iPad apps.

There really isn't much difference between the two: they both run iOS and have access to the exact same frameworks. But the iPad has a much bigger screen (768×1024 points for regular iPads, 1024×1366 points for the 12.9-inch iPad Pro) and that makes all the difference.

In this section you'll make the app *universal* so that it runs on both the iPhone and the iPad. You are not required to always make your apps universal; it is possible to make apps that run only on the iPad and not on the iPhone.

➤ Go to the **Project Settings** screen and select the StoreSearch target.

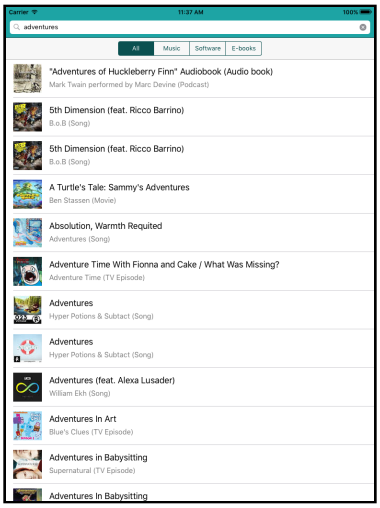
In the **General** tab under **Deployment Info** there is a setting for **Devices**. It is currently set to iPhone, but change it to **Universal**.



Making the app universal

That's enough to make the app run on the iPad.

➤ Choose one of the **iPad** Simulators and run the app. Be aware that the iPad Simulator is huge so you may need to press ⌘**+3** or even ⌘**+4** to make it fit on your computer, especially if you don't have a Retina screen.



StoreSearch in the iPad Simulator

This works but simply blowing up the interface to iPad size is not taking advantage of all the extra space the bigger screen offers. So instead you're going to use some of the special features that UIKit has to offer on the iPad, such as the split-view controller and popovers.

The split-view controller

On the iPhone, a view controller manages the whole screen, although you've seen that you can embed view controllers inside another.

On iPad it is common for view controllers to manage just a section of the screen, because the display is so much bigger and often you will want to combine different types of content in the same screen.

A good example of this is the split-view controller. It has two panes, a big one and a smaller one.

The smaller pane is on the left (the "master" pane) and usually contains a list of items. The right pane (the "detail" pane) shows more information about the thing you have selected in the master list. Each pane has its own view controller.

If you've used an iPad before then you've seen the split-view controller in action because it's used in many of the standard apps such as Mail and Settings.

The split-view controller in landscape and portrait orientations

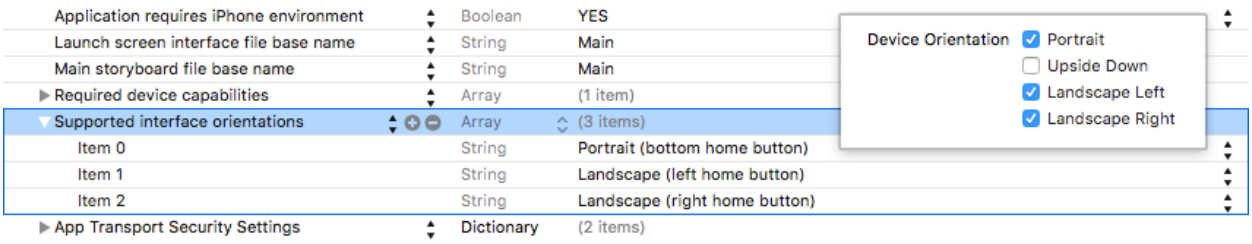
If the iPad is in landscape, the split-view controller has enough room to show both panes at the same time. However, in portrait mode only the detail view controller is visible and the app provides a button that will slide the master pane into view. (You can also swipe the screen to reveal and hide it.)

In this section you'll convert the app to use such a split-view controller. This has some consequences for the organization of the user interface.

Because the iPad has different dimensions from the iPhone it will also be used in different ways. Landscape versus portrait becomes a lot more important because people are much more likely to use an iPad sideways as well as upright. Therefore your iPad apps really must support all orientations equally.

This implies that an iPad app shouldn't make landscape show a completely different UI than portrait, so what you did with the iPhone version of the app won't fly on the iPad – you can no longer show the `LandscapeViewController` when the user rotates the device. That feature goes out of the window.

➤ Open **Info.plist**. There is a **Supported interface orientations** field with three items. This corresponds to the Device Orientation checkboxes under Deployment Info in the Project Settings screen.



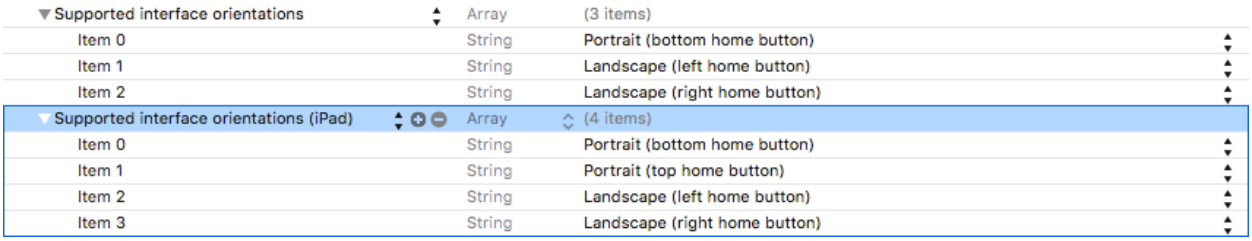
The supported device orientations in Info.plist

The iPad can have its own supported orientations. On the iPhone you usually don't want to enable Upside Down but on the iPad you do.

For some reason, Xcode 8 does not let you change the iPad-specific orientations in the Deployment Info screen so you'll have to add them to Info.plist by hand.

➤ Right-click and choose **Add Row** from the menu. From the list that pops up choose **Supported interface orientations (iPad)**. This already has one row for "Portrait (bottom home button)".

Add three more rows to this setting so that it looks like this:



Adding the supported interface orientations for iPad

All right, that takes care of the orientations. Run the app on the iPad simulator and verify that the app always rotates so that the search bar is on top, no matter what orientation you put the iPad in.

Let's put that split-view controller into the app.

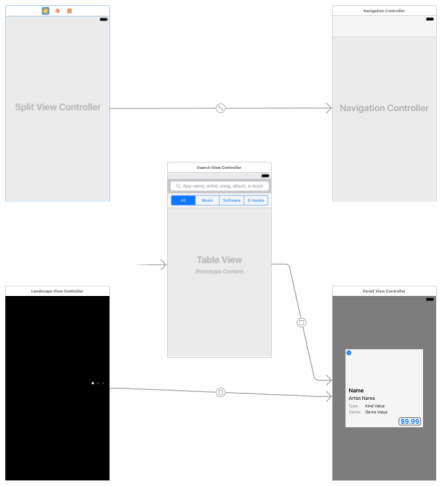
Thanks to universal storyboards you can simply add a Split View Controller object to the storyboard. The split-view is only visible on the iPad; on the iPhone it stays hidden.

This is a lot simpler than in previous iOS versions where you had to make two different storyboard files, one for the iPhone and one for the iPad. Now you just design your entire UI in single storyboard and it magically works across all device types.

➤ Open **Main.storyboard**. Drag a new **Split View Controller** into the canvas. Put it to the left of the Search scene.

➤ The Split View Controller comes with several scenes attached. Remove the white View Controller. Also remove the one that says Root View Controller. Keep the Navigation Controller.

It takes some creativity to make it all fit nicely on the storyboard. Here's how I arranged it:



The storyboard with the new Split View Controller and Navigation Controller

A split-view controller has a relationship segue with two child view controllers, one for the smaller master pane on the left and one for the bigger detail pane on the right.

The obvious candidate for the master pane is the `SearchViewController`, and the `DetailViewController` will go – where else? – into the detail pane.

➤ Ctrl-drag from the Split View Controller to the Search View Controller. Choose **Relationship Segue – master view controller**.

This puts a new arrow between the split-view and the Search screen. (This arrow used to be connected to the navigation controller.)

You won't put the Detail View Controller directly into the split-view's detail pane. It's better to wrap it inside a Navigation Controller first. That is necessary for portrait mode where you need a button to slide the master pane into view. What better place for this button than a navigation bar?

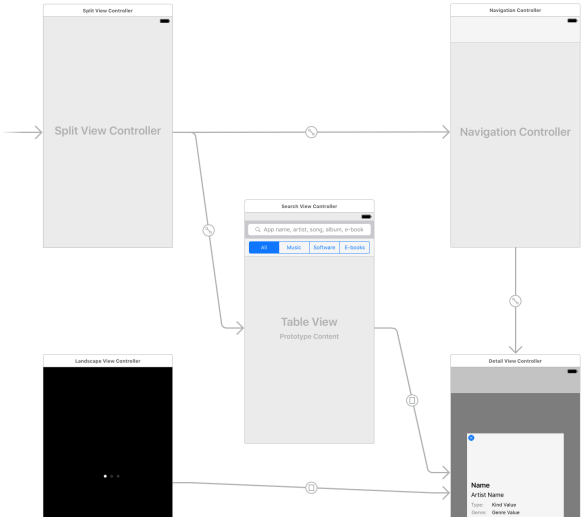
➤ Ctrl-drag from the Split View Controller to the Navigation Controller. Choose **Relationship Segue – detail view controller**.

➤ Ctrl-drag from the Navigation Controller to the Detail View Controller. Make this a **Relationship Segue – root view controller**.

The split-view must become the initial view controller so it gets loaded by the storyboard first.

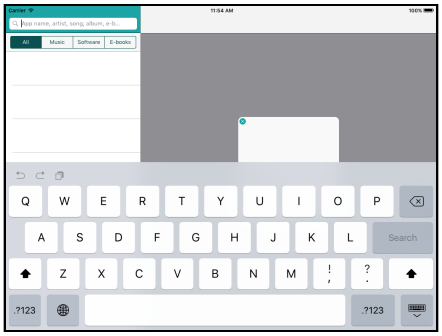
➤ Pick up the arrow that points to Search View Controller and drag it onto the Split View Controller. (You can also check the **Is Initial View Controller** option in the Attributes inspector.)

Now everything is connected:



The master and detail panes are connected to the split-view

And that should be enough to get the app up and running with a split-view:



The app in a split-view controller

It will still take a bit of effort to make everything look good and work well, but this was the first step.

If you play with the app you'll notice that it still uses the logic from the iPhone version and that doesn't always work so well now that the UI sits in a split-view. For example, tapping the price button from the new Detail pane crashes the app…

In the rest of this section you'll be fixing up the app to make sure it doesn't do anything funny on the iPad!

First let's patch up the master pane. It works fine in landscape but in portrait mode it's not visible. You can make it appear by swiping from the left edge of the screen (try it out), but there should really be a button to reveal it as well – the so-called *display mode* button. The split-view controller takes care of most of this logic but you still need to put that button somewhere.

That's why you put `DetailViewController` in a Navigation Controller, so you can add this button – which is a `UIBarButtonItem` – into its navigation bar. (It's not required to use a navigation controller for this. For example, you could also add a toolbar to the `DetailViewController` or use a different button altogether.)

➤ Add the following properties to **AppDelegate.swift**, inside the class:

```
var splitViewController: UISplitViewController {
  return window!.rootViewController as! UISplitViewController
}

var searchViewController: SearchViewController {
  return splitViewController.viewControllers.first
                                          as! SearchViewController
}

var detailNavigationController: UINavigationController {
  return splitViewController.viewControllers.last
                                          as! UINavigationController
}

var detailViewController: DetailViewController {
  return detailNavigationController.topViewController
                                          as! DetailViewController
}
```

These four computed properties refer to the various view controllers in the app:

- `splitViewController`: the top-level view controller
- `searchViewController`: the Search screen in the master pane of the split-view
- `detailNavigationController`: the `UINavigationController` in the detail pane of the split-view
- `detailViewController`: the Detail screen inside the `UINavigationController`

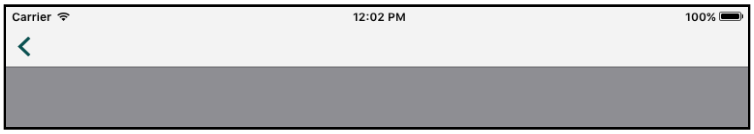
By making properties for these view controllers you can easily refer to them without having to go digging through the hierarchy like you did in the previous tutorials.

➤ Add the following line to `application(didFinishLaunchingWithOptions)`:

```
detailViewController.navigationItem.leftBarButtonItem =
                                splitViewController.displayModeButtonItem
```

This looks up the Detail screen and puts a button into its navigation item for switching between the split-view display modes. Because the `DetailViewController` is embedded in a `UINavigationController`, this button will automatically end up in the navigation bar.

If you run the app now, all you get is a back arrow (in portrait):



The display mode button

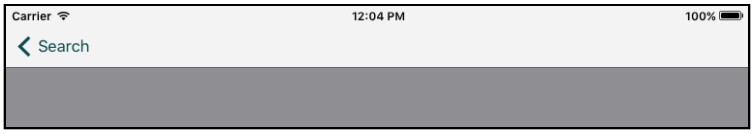
It would be better if this back button said "Search". You can fix that by giving the view controller from the master pane a title.

➤ In **SearchViewController.swift**, add the following line to `viewDidLoad()`:

```
title = NSLocalizedString("Search", comment: "Split-view master button")
```

Of course you're using `NSLocalizedString()` because this is text that appears to the user. Hint: the Dutch translation is "Zoeken".

➤ Run the app and now you should have a proper button for bringing up the master pane in portrait:



The display mode button has a title

Exercise. On the iPad flipping to landscape doesn't bring up the special Landscape View Controller anymore. That's good because we don't want to use it in the iPad version of the app, but you haven't changed anything in the code. Can you explain what stops the landscape view from appearing? ■

Answer: The clue is in `SearchViewController`'s `willTransition()`. This shows the landscape view when the new vertical size class becomes *compact*. But on the iPad both the horizontal and vertical size class are always *regular*, regardless of the device orientation. As a result, nothing happens upon rotation.

Improving the detail pane

The detail pane needs some more work – it just doesn't look very good yet. Also, tapping a row in the search results should fill in the split-view's detail pane, not bring up a new pop-up.

You're using `DetailViewController` for both purposes (pop-up and detail pane), so let's give it a boolean that determines how it should behave. On the iPhone it will be a pop-up; on the iPad it will not.

➤ Add the following instance variable to **DetailViewController.swift**:

```
var isPopUp = false
```

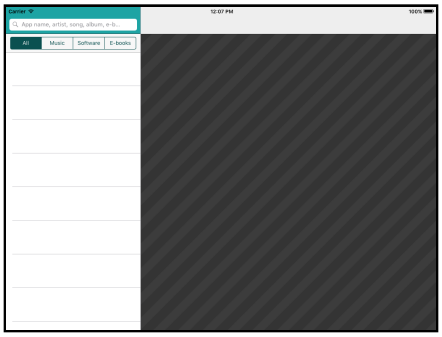
➤ Add the following lines to its `viewDidLoad()` method:

```
if isPopUp {
  let gestureRecognizer = UITapGestureRecognizer(target: self,
                                                 action: #selector(close))
  gestureRecognizer.cancelsTouchesInView = false
  gestureRecognizer.delegate = self
  view.addGestureRecognizer(gestureRecognizer)

  view.backgroundColor = UIColor.clear
} else {
  view.backgroundColor = UIColor(patternImage:
                                 UIImage(named: "LandscapeBackground")!)
  popupView.isHidden = true
}
```

You're supposed to move the code that adds the gesture recognizer into the `if isPopUp` clause, so that tapping the background has no effect on the iPad. Likewise for the line that sets the background color to `clearColor`.

This always hides the labels until a `SearchResult` is selected in the table view. The background gets a pattern image to make things look a little nicer (it's the same image you used with the landscape view on the iPhone).



Making the detail pane look better

Initially this means the `DetailViewController` doesn't show anything (except the patterned background), so you will need to make `SearchViewController` tell the `DetailViewController` that a new `SearchResult` has been selected.

Previously, `SearchViewController` created a new instance of `DetailViewController` every time you tapped a row but now it will need to use the existing instance from the split-view's detail pane instead. But how does the `SearchViewController` know what that instance is?

You will have to give it a reference to the `DetailViewController`. A good place for that is in `AppDelegate` where you create those instances.

➤ Add the following line to `application(didFinishLaunchingWithOptions)`:

```
searchViewController.splitViewDetail = detailViewController
```

This won't work as-is because `SearchViewController` doesn't have an instance variable named `splitViewDetail` yet.

➤ Add this new property to **SearchViewController.swift**:

```
weak var splitViewDetail: DetailViewController?
```

Notice that you make this property `weak`. The `SearchViewController` isn't responsible for keeping the `DetailViewController` alive (the split-view controller is). It would work fine without `weak` but specifying it makes the relationship clearer.

The variable is an optional because it will be `nil` when the app runs on an iPhone.

➤ To change what happens when the user taps a search result on the iPad, replace `tableView(didSelectRowAt)` with:

```
func tableView(_ tableView: UITableView,
               didSelectRowAt indexPath: IndexPath) {
  searchBar.resignFirstResponder()

  if view.window!.rootViewController!.traitCollection
                                     .horizontalSizeClass == .compact {
    tableView.deselectRow(at: indexPath, animated: true)
    performSegue(withIdentifier: "ShowDetail", sender: indexPath)

  } else {
    if case .results(let list) = search.state {
      splitViewDetail?.searchResult = list[indexPath.row]
    }
  }
}
```

On the iPhone this still does the same as before (pop up a new Detail screen), but on the iPad it assigns the `SearchResult` object to the existing `DetailViewController` that lives in the detail pane.

Note: To determine whether the app runs on an iPhone versus an iPad, you're looking at the horizontal size class of the window's root view controller (which

is the `UISplitViewController`).

On the iPhone the horizontal size class is always *compact* (with the exception of the 6 Plus and 6s Plus, more about that shortly). On the iPad it is always *regular*.

The reason you're looking at the size class from the root view controller and not `SearchViewController` is that the latter's size class is always horizontally *compact*, even on iPad, because it sits inside the split-view's master pane.

These changes by themselves don't update the contents of the labels in the `DetailViewController` yet, so let's make that happen.

The ideal place is in a *property observer* on the `searchResult` variable. After all, the user interface needs to be updated right after you put a new `SearchResult` object into this variable.

➤ Change the declaration of `searchResult` in **DetailViewController.swift**:

```
var searchResult: SearchResult! {
  didSet {
    if isViewLoaded {
      updateUI()
    }
  }
}
```

You've seen this pattern a few times before. You provide a `didSet` observer to perform certain functionality when the value of a property changes. After `searchResult` has changed, you call the `updateUI()` method to set the text on the labels.

Notice that you first check whether the controller's view is already loaded. It's possible that `searchResult` is given an object when the `DetailViewController` hasn't loaded its view yet – which is exactly what happens in the iPhone version of the app.

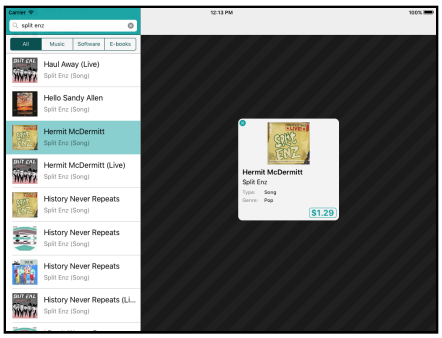
In that case you don't want to call `updateUI()` as there is no user interface yet to update. The `isViewLoaded` check ensures this property observer only gets used when on an iPad.

➤ Add the following line to the bottom of `updateUI()`:

```
popupView.isHidden = false
```

This makes the view visible when on the iPad (recall that in `viewDidLoad()` you hid the pop-up because there was nothing to show yet).

➤ Run the app. Now the detail pane should show details about the selected search result. Notice that the row in the table stays selected as well.



The detail pane shows additional info about the selected item

One small problem: the Detail pop-up no longer works properly on the iPhone because `isPopUp` is always false (try it out, it's hilarious).

➤ In `prepare(for:sender:)` in **SearchViewController.swift**, add the line:

```
detailViewController.isPopUp = true
```

➤ Do the same thing in **LandscapeViewController.swift**. Verify that the Detail screen works properly in all situations.

It would be nice if the app shows its name in the big navigation bar on top of the detail pane. Currently all that space seems wasted. Ideally, this would use the localized name of the app.

You could use `NSLocalizedString()` and put the name into the Localizable.strings files, but considering that you already put the localized app name in InfoPlist.strings it would be handy if you could use that. As it turns out, you can.

➤ In **DetailViewController.swift**, add this line to the `else` clause in `viewDidLoad()`:

```
if let displayName = Bundle.main.
            localizedInfoDictionary?["CFBundleDisplayName"] as? String {
  title = displayName
}
```

The `title` property is used by the `UINavigationController` to put the title text in the navigation bar. You set it to the value of the `CFBundleDisplayName` setting from the localized version of Info.plist, i.e. the translations from InfoPlist.strings.

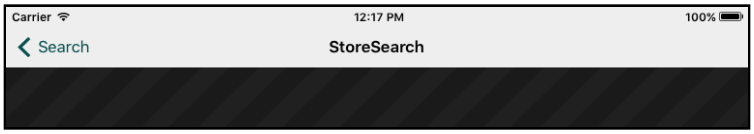
Because `NSBundle`'s `localizedInfoDictionary` can be `nil` you need to unwrap it. The value stored under the `"CFBundleDisplayName"` key may also be `nil`. And finally, the `as?` cast to turn the value to a `String` can also potentially fail. If you're counting along, that is three things that can go wrong in this line of code.

That's why it's called *optional chaining*: you can check a chain of optionals in a single statement. If any of them is `nil`, the code inside the `if` is skipped. That's a lot shorter than writing three separate if-statements!

If you were to run the app right now, no title would show up still (unless you have the Simulator in Dutch) because you did not actually put a translation for `CFBundleDisplayName` in the English version of InfoPlist.strings.

➤ Add the following line to **InfoPlist.strings (Base)**:

```
CFBundleDisplayName = "StoreSearch";
```



That's a good-looking title

There are a few other small improvements to make. On the iPhone it made sense to give the search bar the input focus so the keyboard appeared immediately after launching the app. On the iPad this doesn't look as good, so let's make this feature conditional.

➤ In the `viewDidLoad()` method from **SearchViewController.swift**, put the call to `becomeFirstResponder()` in an if-statement:

```
if UIDevice.current.userInterfaceIdiom != .pad {
  searchBar.becomeFirstResponder()
}
```

To figure out whether the app is running on the iPhone or on the iPad, you look at the current `userInterfaceIdiom`. This is either `.pad` or `.phone` – an iPod touch counts as a phone in this case.

The master pane needs some tweaking also, especially in portrait. After you tap a search result, the master pane stays visible and obscures about half of the detail pane. It would be better to hide the master pane when the user makes a selection.

➤ Add the following method to **SearchViewController.swift**:

```
func hideMasterPane() {
  UIView.animate(withDuration: 0.25, animations: {
    self.splitViewController!.preferredDisplayMode = .primaryHidden
  }, completion: { _ in
    self.splitViewController!.preferredDisplayMode = .automatic
  })
}
```

Every view controller has a built-in `splitViewController` property that is non-`nil` if the view controller is currently inside a `UISplitViewController`.

You can tell the split-view to change its display mode to `.primaryHidden` to hide the master pane. You do this in an animation block, so the master pane disappears with a smooth animation.

The trick is to restore the preferred display mode to `.automatic` after the animation completes, otherwise the master pane stays hidden even in landscape!

➤ Add the following lines to `tableView(didSelectRowAt)` in the `else` clause, right below the `if case .results` statement:

```
if splitViewController!.displayMode != .allVisible {
  hideMasterPane()
}
```

The `.allVisible` mode only applies in landscape, so this says, "if the split-view is not in landscape, hide the master pane when a row gets tapped."

➤ Try it out. Put the iPad in portrait, do a search, and tap a row. Now the master pane will slide out when you tap a row in the table.

Congrats! You have successfully repurposed the Detail pop-up to also work as the detail pane of a split-view controller. Whether this is possible in your own apps depends on how different you want the user interfaces of the iPhone and iPad versions to be.

Often you'll find that the iPad user interface for your app is different enough from the iPhone's that you will have to make all new view controllers with some duplicated logic. If you're lucky you may be able to use the same view controllers for both versions of the app but often that is more trouble than it's worth.

The Apple Developer Forums

When I wrote this chapter, how to hide the master pane was not explained anywhere in the official `UISplitViewController` documentation and I had trouble getting it to work properly.

Desperate, I turned to the Apple Developer Forums and asked my question there. Within a few hours I received a reply from a fellow developer who ran into the same problem and who found a solution – thanks, user "timac"!

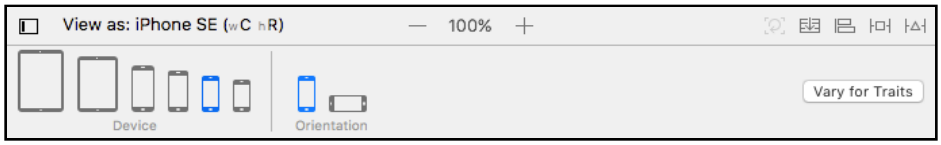
So if you're stuck, don't forget to look at the Apple Developer Forums for a solution: https://forums.developer.apple.com

Size classes in the storyboard

Even though you've placed the existing `DetailViewController` in the detail pane, the app is not using all that extra iPad space effectively. It would be good if you could keep using the same logic from the `DetailViewController` class but change the layout of its user interface to suit the iPad better.

If you like suffering, you could do `if UIDevice.current.userInterfaceIdiom == .pad` in `viewDidLoad()` and move all the labels around programmatically, but this is exactly the sort of thing size classes were invented for.

➤ Open **Main.storyboard** and take a look at the **View as:** pane.

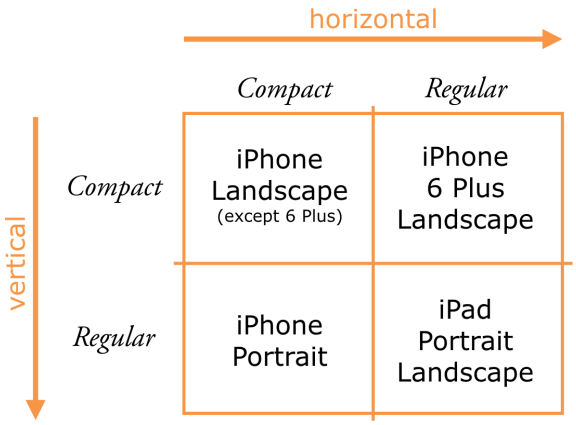


Size classes in the View as: pane

Notice how it says **iPhone SE (wC hR)**? The **wC** and **hR** are the size class for this particular device: the size class for the width is *compact* (wC), and the size class for the height is *regular* (hR).

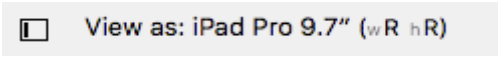
Recall that there are two possible size classes, *compact* and *regular*, and that you can assign one of these values to the horizontal axis (Width) and one to the vertical axis (Height).

Here is the diagram again:



Horizontal and vertical size classes

➤ Use the **View as:** pane to switch to **iPad Pro (9.7")**. Not only are the view controllers larger now, but you'll see the size class has changed to **wR hR**, or *regular* in both width and height.



The size classes for the iPad

We want to make the Detail pop-up bigger when the app runs on the iPad. However, if you make any edits to the storyboard right now, these edits will also affect the design of the app in iPhone mode. Fortunately, there is a way to make edits that apply to a specific size class only.

You can tell Interface Builder that you only want to change the layout for the *regular* width size class (**wR**), but leave *compact* width alone (**wC**). Now those edits will only affect the appearance of the app on the iPad.

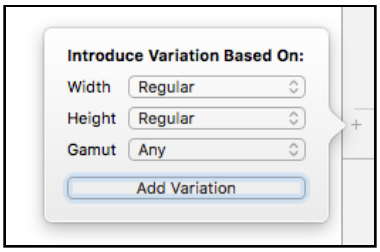
For example, the Detail pane doesn't need a close button on the iPad. It is not a pop-up so there's no reason to dismiss it. Let's remove that button from the storyboard.

➤ Select the **Close Button**. Go to the Attributes inspector and scroll all the way to the bottom, to the **Installed** option.

The installed checkbox

This option lets you remove a view from a specific size class, while leaving it visible in other size classes.

➤ Click the tiny **+** button to the left of Installed. This brings up a menu. Choose **Width: Regular**, **Height: Regular** and click on **Add Variation**:



Adding a variation for the regular, regular size class

This adds a new line with a second Installed checkbox:

The option can be changed on a per-size class basis

➤ Uncheck Installed for **wR hR**. Now the Close Button disappears from the scene (if the storyboard is in iPad mode).

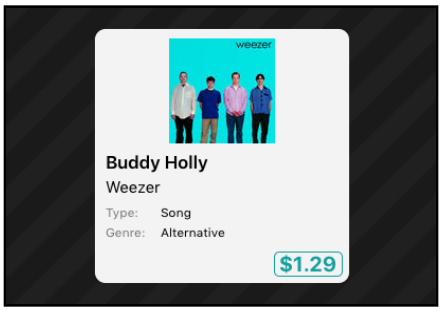
The Close Button still exists, but it is not installed in this size class. You can see the button in the outline pane but it is grayed out.

The Close Button is still present but grayed out

➤ Use the **View as:** panel to switch back to **iPhone SE**.

Notice how the Close Button is back in its original position. You've only removed it from the storyboard design for the iPad. That's the power of universal storyboards and size classes.

➤ Run the app and you'll see that the close button really is gone on the iPad:



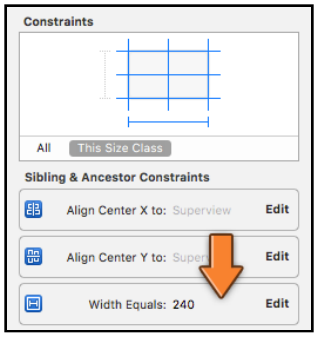
No more close button in the top-left corner

Using the same principle you can change the layout of the Detail screen to be completely different between the iPhone and iPad versions.

➤ In the storyboard, switch to the **iPad Pro** layout again.

You will now change the size of the Width constraint for the Pop-up View from 240 to 500 points.

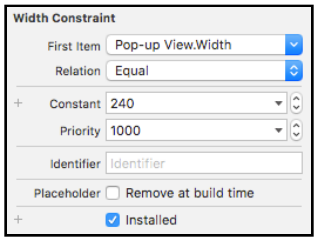
➤ Select the **Pop-up View** and go to the **Size inspector**. The **Constraints** section shows the constraints for this view:



The Size inspector lists the constraints for the selected view

The **Width Equals: 240** constraint has an **Edit** button. If you click that, a popup appears that lets you change the width. However, that will change this constraint for *all* size classes. You want to change it for the iPad only. Instead, do the following.

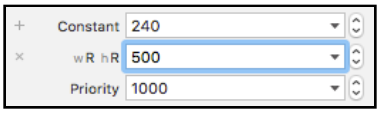
➤ Double-click **Width Equals: 240**. This brings up the **Size inspector** for just that constraint:



The Size inspector for the Width constraint

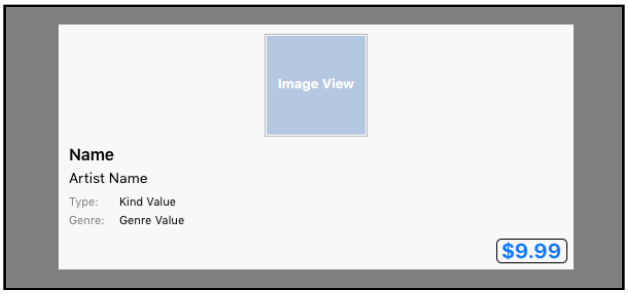
(If you just type in a new value for Constant, the constraint will become larger for all size classes again.)

➤ Click the **+** button next to Constant. In the popup choose **Width: Regular**, **Height: Regular** and click **Add Variation**. This adds a second row. Type **500** into the new **wR hR** field.



Adding a size class variation for the Constant

Now the Pop-up View is a lot wider. Next up you'll rearrange and resize the labels to take advantage of the extra space.



The Pop-up View after changing the Width constraint

➤ In a similar manner, change the **Width** and **Height** constraints of the **Image View** to **180**.

➤ Select the **Vertical Space** constraint between the **Name** label and the **Image View** and go to its **Size inspector**. Add a new variation for Constant and type **28** into the **wR hR** field.

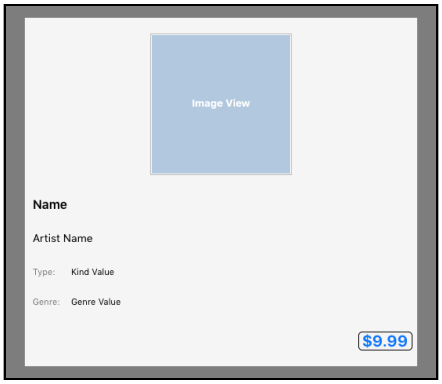
➤ Repeat this procedure for the other **Vertical Space** constraints. Each time use the **+** button to add a new rule for **Width: Regular**, **Height: Regular**, and make the new Constant 20 points taller than standard one.

Remember, if the constraints are difficult to pinpoint, then select the view they're attached to instead and use the Size inspector to find the actual constraints.

➤ Make the **Vertical Space** at the top of the **Image View** 20 points.

➤ And finally, put the **$9.99 button** at 20 points from the sizes instead of 6.

You should end up with something that looks like this:

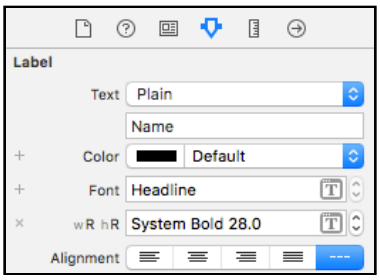


The Pop-up View after changing the vertical spaces

Just to double-check, switch back to iPhone SE and make sure that the Detail pane is restored to its original dimensions. If not, then you may have changed one of the original constraints instead of making a variation for the iPad's size class.

In the iPad's version of the Detail pane, the text is now tiny compared to the pop-up background, so let's change the fonts. That works in the same fashion: you add a customization for this size class with the **+** button, then change the property. (Any attribute that has a small + in front of it can be customized for different size classes.)

➤ Select the **Name** label. In the **Attributes inspector** click the **+** in front of **Font**. Choose the **System Bold** font, size **28**.



Adding a size class variation for the label's font

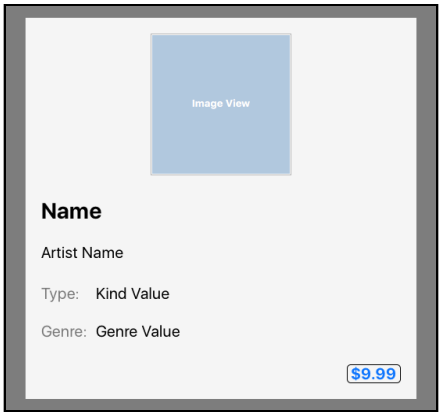
If orange lines appear, use the Resolve Auto Layout Issues menu to update the positions and sizes of the views so they correspond with the constraints again (tip: choose Update Frames from the "All Views" section).

➤ Change the font of the other labels to **System**, size **20**. You can do this in one go by making a multiple-selection.

I'm not entirely happy with the margins for the labels yet.

➤ Change all the "leading" **Horizontal Space** constraints to **20** for this size class.

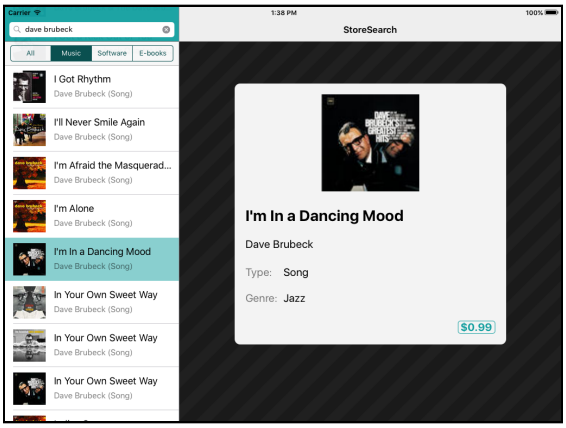
The final layout should look like this:



The layout for the Pop-up View on iPad

Switch back to iPhone SE to make sure all the constraints are still correct there.

➤ Run the app and you should have a much bigger detail view:



The iPad now uses different constraints for the detail pane

Exercise. The first time the detail pane shows its contents they appear quite abruptly because you simply set the `isHidden` property of `popupView` to `false`, which causes it to appear instantaneously. See if you can make it show up using a cool animation. ■

➤ This is probably a good time to try the app on the iPhone again. The changes you've made should be compatible with the iPhone version, but it's smart to make sure.

If you're satisfied everything works as it should, then commit the changes.

Slide over and split-screen on iPad

iOS has a very handy split-screen feature that lets you run two apps side-by-side. It only works on the higher-end iPads such as the iPad Air 2 and iPad Pro. Because you used size classes to build the app's user interface, split-screen support works flawlessly.

Try it out: run the app on the iPad Air 2 or iPad Pro simulator. Swipe from the right edge of the screen towards the middle. This opens a panel that lets you choose another app to overlay on top of StoreSearch. To put the two apps side-by-side, drag the divider bar to the middle of the screen. Thanks to size classes, the layout of StoreSearch automatically adapts to the allotted space.

The **View as:** panel has a button **Vary for Traits**. You can use this to change how a view controller acts when it is part of such a split screen.

Your own popover

Anyone who has ever used an iPad before is no doubt familiar with popovers, the floating panels that appear when you tap a button in a navigation bar or toolbar.

They are a very handy UI element.

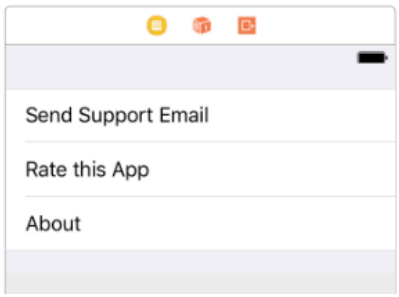
A popover is nothing more than a view controller that is presented in a special way. In this section you'll create a popover for a simple menu.

➤ In the storyboard, first switch back to **iPhone SE** because in iPad mode the view controllers are huge and we can use the extra space to work with.

➤ Drag a new **Table View Controller** into the canvas and place it next to the Detail screen.

➤ Change the table view to **Grouped** style and give it **Static Cells**.

➤ Add these rows (change the cell style to **Basic**):

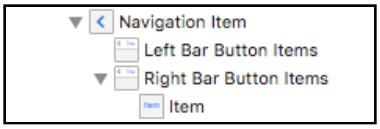


The design for the new table view controller

This just puts three items in the table. You will only do something with the first one in this tutorial. Feel free to implement the functionality of the other two by yourself.

To show this view controller inside a popover, you first have to add a button to the navigation bar so that there is something to trigger the popover from.

➤ From the Object Library drag a new **Bar Button Item** into the **Detail View Controller**'s **Navigation Item**. You can find this in the outline pane. Make sure the Bar Button Item is in the Right Bar Button Items group.

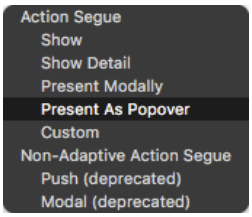


The new bar button item in the Navigation Item

➤ Change the bar button's **System Item** to **Action**.

This button won't show up on the iPhone because there the Detail pop-up doesn't sit in a navigation controller.

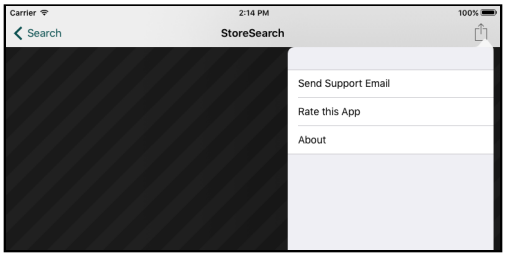
➤ Ctrl-drag from the bar button (in the outline pane) to the Table View Controller to make segue. Choose segue type **Action Segue – Present As Popover**.



The new bar button item in the Navigation Item

➤ Give the segue the identifier **ShowMenu**.

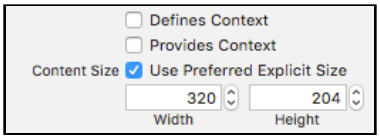
If you run the app and press the menu button, the app looks like this:



That menu is a bit too tall

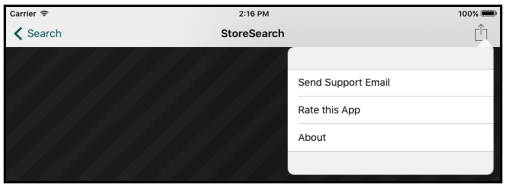
The popover doesn't really know how big its content view controller is, so it just picks a size. That's ugly, but you can tell it how big the view controller should be with the *preferred content size* property.

➤ In the **Attributes inspector** for the **Table View Controller**, in the **Content Size** boxes type Width: 320, Height: 204.



Changing the preferred width and height of the popover

Now the size of the menu popover looks a lot more appropriate:

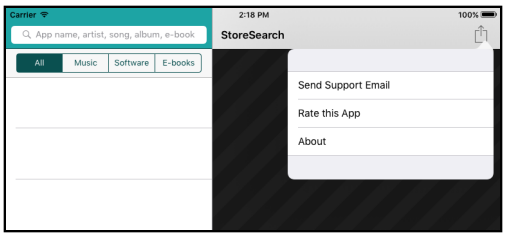


The menu popover with a size that fits

When a popover is visible, all other controls on the screen become inactive. The user has to tap outside of the popover to dismiss it before she can use the rest of

the screen again (you can make exceptions to this by setting the popover's `passthroughViews` property).

While the menu popover is visible, the other bar button (Search) is still active as well if you're in portrait mode. This can create a situation where two popovers are open at the same time:



Both popovers are visible

That is a violation of the rules from Apple's Human Interface Guidelines, also known as the "HIG". The folks at Apple don't like it when apps show more than one popover at a time, probably because it is confusing to the user which one requires input. The app will be rejected from the App Store for this, so you have to make sure this situation cannot happen.

The scenario you need to handle is when the user first opens the menu popover followed by a tap on the Search button. To fix this issue, you need to know when the Search button is pressed and the master pane becomes visible, so you can hide the menu popover.

Wouldn't you know it... of course there is a delegate method for that.

➤ Add the following extension to the bottom of **AppDelegate.swift**:

```
extension AppDelegate: UISplitViewControllerDelegate {
  func splitViewController(_ svc: UISplitViewController,
          willChangeTo displayMode: UISplitViewControllerDisplayMode) {
    print(#function)
    if displayMode == .primaryOverlay {
      svc.dismiss(animated: true, completion: nil)
    }
  }
}
```

This method dismisses any presented view controller – that would be the popover – if the display mode changes to `.primaryOverlay`, in other words if the master pane becomes visible.

> **Note:** The line `print(#function)` is a useful tip for debugging. This prints out the name of the current function or method to the Xcode debug pane. That quickly tells you when a certain method is being called.

You still need to tell the split-view controller that `AppDelegate` is its delegate.

➤ Add the following line to `application(didFinishLaunchingWithOptions)`:

```
splitViewController.delegate = self
```

And that should do it! Try having both the master pane and the popover in portrait mode. Ten bucks says you can't!

Sending email from within the app

Now let's make the "Send Support Email" menu option work. Letting users send an email from within your app is pretty easy.

iOS provides the `MFMailComposeViewController` class that takes care of everything for you. It lets the user type an email and then sends the email using the mail account that is set up on the device.

All you have to do is create an `MFMailComposeViewController` object and present it on the screen.

The question is: who will be responsible for this mail compose controller? It can't be the popover because that view controller will be deallocated once the popover goes away.

Instead, you will let the `DetailViewController` handle the sending of the email, mainly because this is the screen that brings up the popover in the first place (through the segue from its bar button item). `DetailViewController` is the only object that knows anything about the popover.

To make this work, you'll create a new class `MenuViewController` for the popover, give it a delegate protocol, and have `DetailViewController` implement those delegate methods.

➤ Add a new file to the project using the **Cocoa Touch Class** template. Name it **MenuViewController**, subclass of **UITableViewController**.

➤ Remove all the data source methods from this file because you don't need those for a table view with static cells.

➤ In the storyboard, change the **Class** of the popover's table view controller to **MenuViewController**.

➤ Add a new protocol to **MenuViewController.swift** (outside the class):

```
protocol MenuViewControllerDelegate: class {
  func menuViewControllerSendSupportEmail(_ controller:
                                          MenuViewController)
}
```

➤ Also add a property for this protocol inside the class:

```
weak var delegate: MenuViewControllerDelegate?
```

Like all delegate properties this is weak because you don't want `MenuViewController` to "own" the object that implements the delegate methods.

➤ Finally, add `tableView(didSelectRowAt)` to handle taps on the rows from the table view:

```
override func tableView(_ tableView: UITableView,
                        didSelectRowAt indexPath: IndexPath) {
  tableView.deselectRow(at: indexPath, animated: true)

  if indexPath.row == 0 {
    delegate?.menuViewControllerSendSupportEmail(self)
  }
}
```

Now you'll have to make `DetailViewController` the delegate for this menu popover. Of course that happens in *prepare-for-segue*.

➤ Add the `prepare(for:sender:)` method to **DetailViewController.swift**:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
  if segue.identifier == "ShowMenu" {
    let controller = segue.destination as! MenuViewController
    controller.delegate = self
  }
}
```

This tells the `MenuViewController` object who the `DetailViewController` is.

➤ Also add the following extension to the bottom of the source file:

```
extension DetailViewController: MenuViewControllerDelegate {
  func menuViewControllerSendSupportEmail(_: MenuViewController) {
  }
}
```

It doesn't do anything yet but the app should compile without errors again.

Run the app and tap Send Support Email. Notice how the popover doesn't disappear yet. You have to manually dismiss it before you can show the mail compose sheet.

➤ The `MFMailComposeViewController` lives in the `MessageUI` framework, so import that in **DetailViewController.swift**:

```
import MessageUI
```

➤ Then add the following code into `menuViewControllerSendSupportEmail()`:

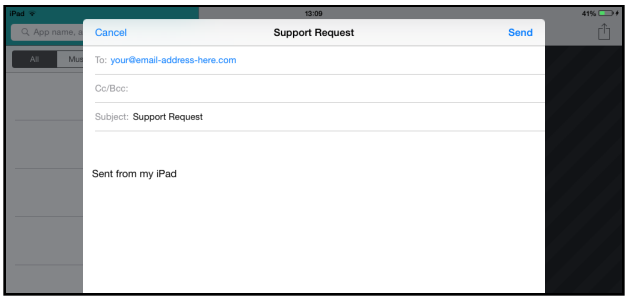
```
dismiss(animated: true) {
  if MFMailComposeViewController.canSendMail() {
    let controller = MFMailComposeViewController()
    controller.setSubject(NSLocalizedString("Support Request",
                                            comment: "Email subject"))
    controller.setToRecipients(["your@email-address-here.com"])
    self.present(controller, animated: true, completion: nil)
  }
}
```

This first calls `dismiss(animated)` to hide the popover. This method takes a `completion` closure that until now you've always left `nil`. Here you do give it a closure – using trailing syntax – that brings up the `MFMailComposeViewController` after the popover has faded away.

It's not a good idea to present a new view controller while the previous one is still in the process of being dismissed, which is why you wait to show the mail compose sheet until the popover is done animating.

To use the `MFMailComposeViewController` object, you have to give it the subject of the email and the email address of the recipient. You probably should put your own email address here!

➤ Run the app and pick the Send Support Email menu option. A form slides up the screen that lets you write an email.



The email interface

> **Note:** If you're running the app on a device and you don't see the email form, you may not have set up any email accounts on your device.
>
> If on the Simulator the email form does not respond, then that's caused by a bug in the "MailCompositionService". It has a tendency to crash.

Notice that the Send and Cancel buttons don't actually appear to do anything. That's because you still need to implement the delegate for this screen.

➤ Add a new extension to **DetailViewController.swift**:

```
extension DetailViewController: MFMailComposeViewControllerDelegate {
  func mailComposeController(_ controller: MFMailComposeViewController,
              didFinishWith result: MFMailComposeResult, error: Error?) {
    dismiss(animated: true, completion: nil)
  }
}
```

The `result` parameter says whether the mail could be successfully sent or not. This app doesn't really care about that, but you could show an alert in case of an error if you want. Check the documentation for the possible result codes.

➤ In the `menuViewControllerSendSupportEmail()` method, add the following line:

```
controller.mailComposeDelegate = self
```

➤ Now if you press Cancel or Send, the mail compose sheet gets dismissed.

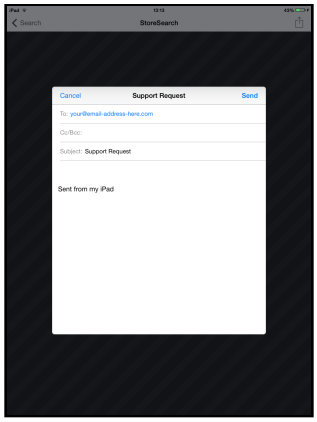
If you're testing on the Simulator, no email actually gets sent out, so don't worry about spamming anyone when you're testing this.

Did you notice that the mail form did not take up the entire space in the screen in landscape, but when you rotate to portrait it does? That is called a **page sheet**. On the iPhone if you presented a modal view controller it always took over the entire screen, but on the iPad you have several options. The page sheet is probably the nicest option for the `MFMailComposeViewController`, but let's experiment with the other ones as well, shall we?

➤ In `menuViewControllerSendSupportEmail()`, add the following line:

```
controller.modalPresentationStyle = .formSheet
```

The `modalPresentationStyle` property determines how a modal view controller is presented on the iPad. You've switched it from the default page sheet to a **form sheet**, which looks like this:



The email interface in a form sheet

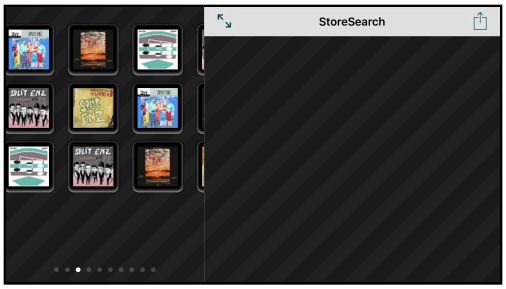
A form sheet is smaller than a page sheet so it always takes up less room than the entire screen. There is also a "full screen" presentation style that always covers the entire screen, even in landscape. Try it out!

Landscape on the iPhone 6s Plus and 7 Plus

The iPhone Plus is a strange beast. It mostly works like any other iPhone but sometimes it gets ideas and pretends to be an iPad.

➤ Run the app on the **iPhone 7 Plus** Simulator, and rotate to landscape.

The app will look something like this:



The landscape screen appears in the split-view's master pane

LOL. The app tries to do both: show the split-view controller and the special landscape view at the same time. Obviously, that's not going to work.

The iPhone 7 Plus, and the older 6s Plus and 6 Plus, is so big that it's almost a small iPad. The designers at Apple decided that in landscape orientation the Plus should behave like an iPad, and therefore it shows the split-view controller.

What's the trick? Size classes, of course. On a landscape iPhone Plus, the horizontal size class is *regular*, not *compact*. (The vertical size class is still compact, just like on the smaller iPhone models.)

To stop the `LandscapeViewController` from showing up, you have to make the rotation logic smarter.

➤ In **SearchViewController**.swift, change `willTransition()` to:

```
override func willTransition(to newCollection: UITraitCollection,
              with coordinator: UIViewControllerTransitionCoordinator) {
  super.willTransition(to: newCollection, with: coordinator)

  let rect = UIScreen.main.bounds
  if (rect.width == 736 && rect.height == 414) ||   // portrait
     (rect.width == 414 && rect.height == 736) {    // landscape
    if presentedViewController != nil {
      dismiss(animated: true, completion: nil)
    }
  } else if UIDevice.current.userInterfaceIdiom != .pad {
```

```
      switch newCollection.verticalSizeClass {
      case .compact:
        showLandscape(with: coordinator)
      case .regular, .unspecified:
        hideLandscape(with: coordinator)
      }
    }
  }
```

The bottom bit of this method is as before; it checks the vertical size class and decides whether to show or hide the `LandscapeViewController`.

You don't want to do this for the iPhone 7 Plus, so you need to detect somehow that the app is running on the Plus. There are a couple of ways you can do this:

- Look at the width and height of the screen. The dimensions of the iPhone 7 Plus are 736 by 414 points.
- Look at the screen scale. Currently the only device with a 3x screen is the Plus. This is not an ideal method because users can enable Display Zoom to get a zoomed-in display with larger text and graphics. That still reports a 3x screen scale but it no longer gives the 6s Plus its own size class. It now acts like other iPhones and the split-view won't appear anymore.
- Look at the hardware machine name of the device. There are APIs for finding this out, but you have to be careful: often one type of iPhone can have multiple model names, depending on the cellular chipset used or other factors.

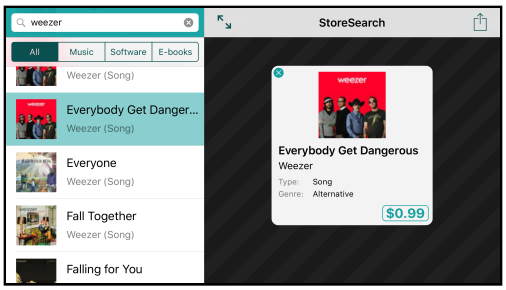
What about the size class? That sounds like it would be the obvious thing to tell the different devices apart. Unfortunately, looking at the size class *doesn't* work.

If the device is in portrait, the 6s Plus has the same size classes as the other iPhone models. In other words, in portrait you can't tell from the size class alone whether the app is running on a Plus or not – only in landscape.

The approach you're using in this app is to look at the screen dimensions. That's the cleanest solution I could find. You need to check for both orientations, because the screen bounds change depending on the orientation of the device.

Once you've detected the app runs on an iPhone 6s Plus or 7 Plus, you no longer show the landscape view. You do dismiss any Detail pop-up that may still be visible before you rotate to landscape.

➤ Try it out. Now the iPhone 7 Plus shows a proper split-view.



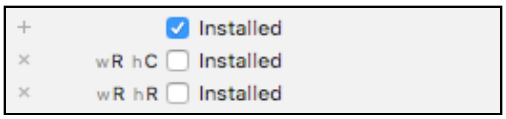
The app on the iPhone 7 Plus with a split-view

Of course the Detail pane now uses the iPhone-size design, not the iPad design.

That's because the size class for `DetailViewController` is now *regular* width, *compact* height. You didn't make a specific design for those size classes, so the app uses the default design.

That's fine for the size of the Detail view, but it does mean the close button is visible again.

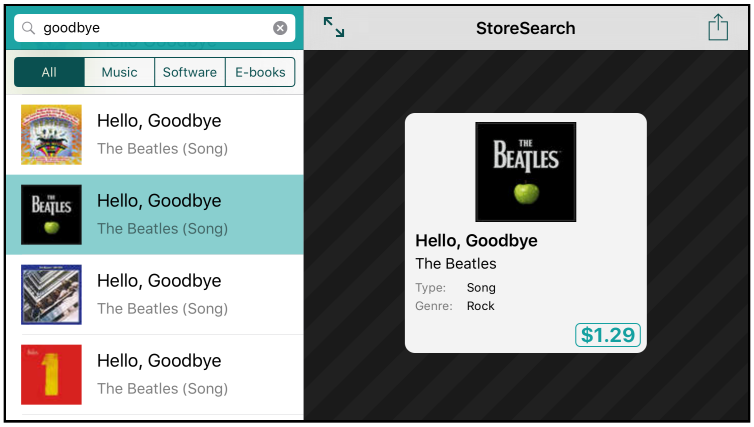
➤ Open the storyboard and select the **Close Button**. In the **Attributes inspector**, add a new row for **Installed** and uncheck it:



Adding a variation for size class width regular, height compact

➤ Select the **Center Y Alignment** constraint on **Pop-up View**. Change its **Constant** to **20**, but only for this size class. This moves the Detail panel down a bit.

(Remember you can preview the effect of these changes by using the **View as:** panel to switch to the iPhone 7 Plus and put it in landscape mode.)



The finished StoreSearch app on the iPhone 6s Plus or 7 Plus

And that's it for the StoreSearch app! Congratulations for making it this far, it has been a long tutorial.

➤ Celebrate by committing the final version of the source code and tagging it v1.0!

You can find the project files for the complete app under **10 - Finished App** in the tutorial's Source Code folder.

What do you do with an app that is finished? Upload it to the App Store, of course! (And with a little luck, make some big bucks…)

Distributing the app

Throughout these tutorials you've probably been testing the apps on the Simulator and occasionally on your device. That's great, but when the app is nearly done you may want to let other people beta test it. You can do this on iOS with so-called **ad hoc** distributions.

Your Developer Program membership allows you to register up to 100 devices with your account and to distribute your apps to the users of those devices, without requiring that they buy the apps from the App Store. You simply build your app in Xcode and send your testers a ZIP file that contains your application bundle and your Ad Hoc Distribution profile. The beta testers can then drag these files into iTunes and sync their iPhones and iPads to install the app.

In this section you'll learn how to make an Ad Hoc distribution for the StoreSearch app. Later on I'll also show you how to submit the app to the App Store, which is a very similar process. (By the way, I'd appreciate it if you don't actually submit the apps from these tutorials. Let's not spam the App Store with dozens of identical "StoreSearch" or "Bull's Eye" apps.)

Join the program

Once you're ready to make your creations available on the App Store, it's time to join the paid Apple Developer Program.

To sign up, go to developer.apple.com/programs/ and click the blue **Enroll** button.

On the sign-up page you'll need to enter your Apple ID. Your developer program membership will be tied to this account. It's OK to use the same Apple ID that you're already using with iTunes and your iPhone, but if you run a business you might want to create a new Apple ID to keep these things separate.

You can enroll as an Individual or as an Organization. There is also an Enterprise program but that's for big companies who will be distributing apps within their own organization only. If you're still in school, the University Program may be worth looking into.

You buy the Developer Program membership from the online Apple Store for your particular country. Once your payment is processed you'll receive an activation code that you use to activate your account.

Signing up is usually pretty quick. In the worst case it may take a few weeks, as Apple will check your credit card details and if they find anything out of the ordinary (such as a misspelled name) your application may run into delays. So make sure to enter your credit card details correctly or you'll be in for an agonizing wait.

If you're signing up as an Organization then you also need to provide a D-U-N-S Number, which is free but may take some time to request. You cannot register as an Organization if you have a single-person business such as a sole proprietorship or DBA ("doing business as"). In that case you need to sign up as an Individual.

You will have to renew your membership every year but if you're serious about developing apps then that $99/year will be worth it.

The distribution profile

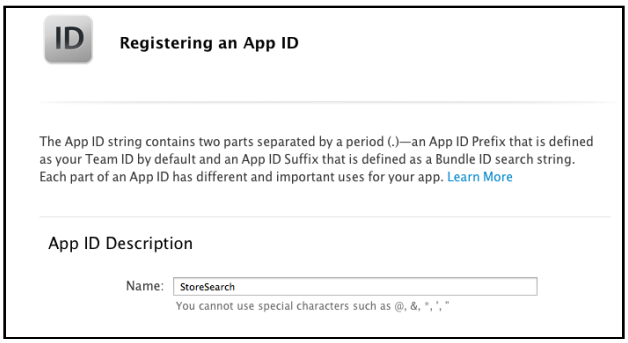
Before you can put your app on a device, it must be signed with your **certificate** and a **provisioning profile**. So far when you've run apps on your device, you have used the Developer Certificate and the Team Provisioning Profile but these are only for development purposes and can only be used from within Xcode.

You probably don't want to send your app's source code to your beta testers, or require them to mess around with Xcode, so you must create a new certificate and profile that are just for distribution.

➤ Open your favorite web browser and surf to the Developer Member Center at https://developer.apple.com/membercenter/. Sign in and go to **Certificates, Identifiers & Profiles**.

> **Note:** Like any piece of software, the Developer Member Center changes every now and then. It's possible that by the time you read this, some of the options are in different places or have different names. The general flow should still be the same, though. And if you really get stuck, online help is usually available.
>
> Tip: If using this website gives you problems such as pages not loading correctly, then try it with Safari. Other browsers sometimes give strange errors.

➤ Click on **App IDs** under **Identifiers** in the sidebar. In the new page that appears, press the **+** button to add a new App ID.



Creating a new App ID

➤ Fill in the **App ID Description** field. This can be anything you want – it's just for usage on the Provisioning Portal.

➤ The **App ID Prefix** field contains the ID for your team. You can leave this as-is.

➤ Under **App ID Suffix**, select **Explicit App ID**. In the **Bundle ID** field you must enter the identifier that you used when you created the Xcode project. For me that is **com.razeware.StoreSearch**.

The Bundle ID must match with the identifier from Xcode

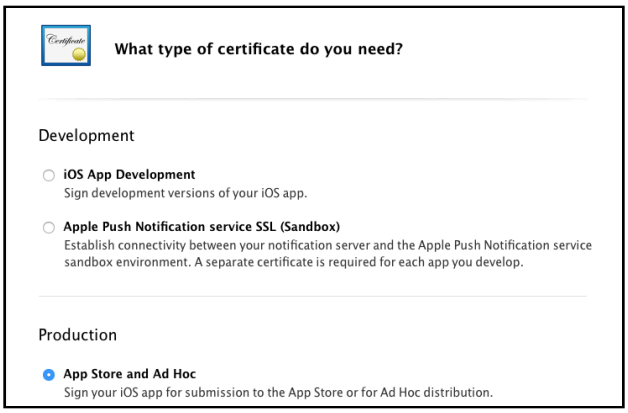
If you want your app to support push notifications, In-App Purchases, or iCloud then you can also configure that here. StoreSearch doesn't need any of that so leave the other fields on the default settings.

➤ Press **Continue** and then **Register** to create the App ID. The portal will now generate the App ID for you and add it to the list.

The full App ID is something like **U89ECKP4Y4.com.yourname.StoreSearch**. That number in front is the ID of your team.

If you do not have a distribution certificate yet, you have to create one.

➤ In the sidebar go to **Certificates**, **Production**. If there is nothing in the list, click the **+** button to create a new certificate.



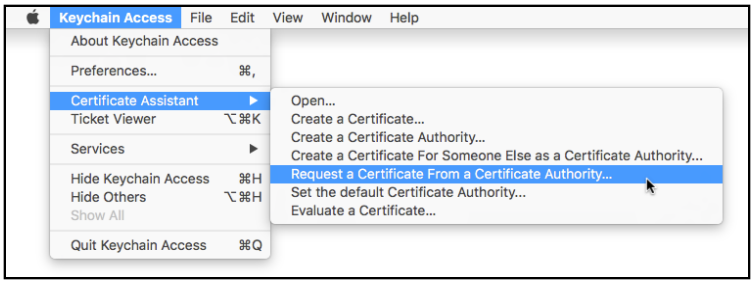
Creating a new distribution certificate

➤ Select the **App Store and Ad Hoc** type, under **Production**. Click **Continue**.

As part of the certificate creation process you need to generate a CSR or Certificate Signing Request. It sounds scary but follow these steps and you'll be fine:

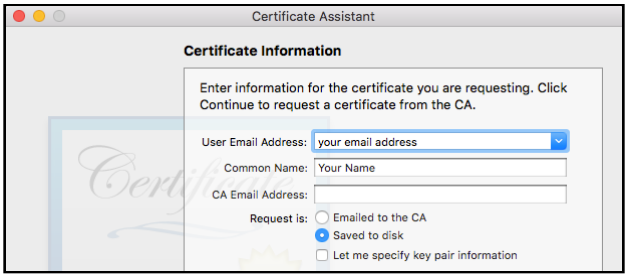
➤ Open the **Keychain Access** app on your Mac (it is in Applications/Utilities).

➤ From the **Keychain Access** menu, choose **Certificate Assistant → Request a Certificate from a Certificate Authority...**:



Using Keychain Access to create a CSR

➤ Fill out the fields in the window that pops up:



Filling out the certificate request

- **User Email Address**: Enter the email address that you used to sign into the Member Center. This is the Apple ID from your Developer Program account.
- **Common Name**: Fill in your name or your company's name.

➤ Check the **Saved to disk** option and press **Continue**. Save the file to your Desktop.

➤ Go back to the web browser and continue to the next step. Upload the **CertificateSigningRequest.certSigningRequest** file you just created and click **Generate**.

After a couple of seconds you should be the owner of a brand new distribution certificate.

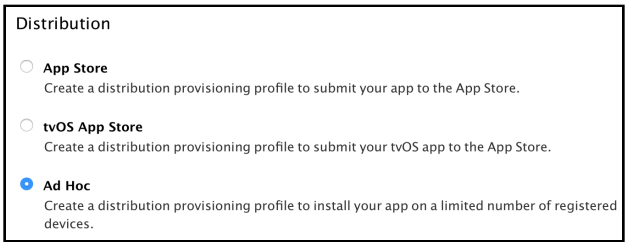
➤ Click the big **Download** button. This saves a file named **ios_distribution.cer** on your computer. Double-click this file to install it. You should be able to see the new certificate in the Keychain Access app under My Certificates.

There's one more thing to do in the Member Center.

➤ In the left-hand menu, under **Provisioning Profiles**, click **Distribution**. This will show your current distribution profiles. (You probably don't have any yet.)

There are two types of distribution profiles: Ad Hoc and App Store. You'll first make an Ad Hoc profile.

➤ Click the **+** button to create a new profile:

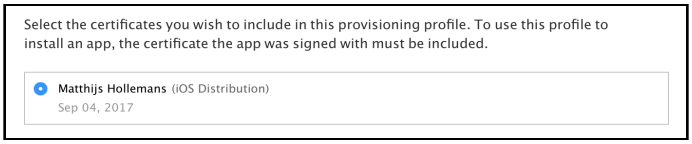


Creating a new provisioning profile for distribution

➤ Select **Ad Hoc** and click **Continue**.

➤ The next step asks you to select an App ID. Pick the App ID that you just created ("StoreSearch").

Now the portal asks you to select the certificate that should be used to create this provisioning profile:

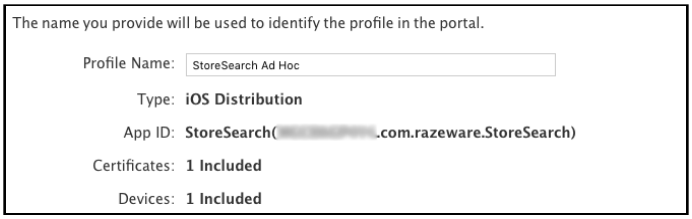


Selecting the certificate

In the next step you need to select the devices for which the provisioning profile is valid. If you're sending the app to beta testers, their devices need to be included in this list. (To add a new device, use the Devices menu option in the portal).

➤ Select your device(s) from the list and click **Continue**.

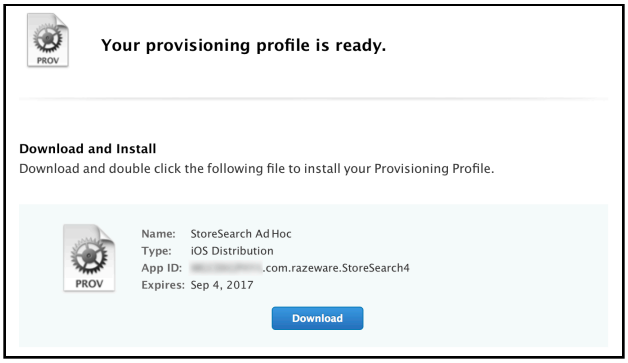
➤ Give the profile a name, for example **StoreSearch Ad Hoc**. Picking a good name is useful for when you have a lot of apps.



Giving the provisioning profile a name

➤ If everything looks OK, click **Generate**.

After a few seconds the provisioning profile is ready for download.



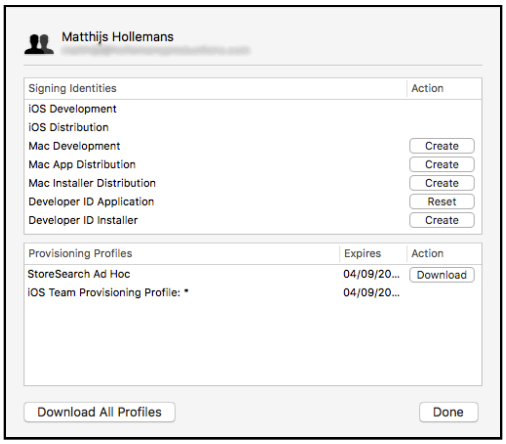
The provisioning profile was successfully created

➤ Click **Download** to download the file **StoreSearch_Ad_Hoc.mobileprovision**. Keep this file safe somewhere; you'll need it later.

➤ Go back to Xcode and open the **Preferences** window. Go to the **Accounts** tab.

If you haven't added your Developer Program account here yet, then click **+** and fill in your Apple ID and password.

➤ Click on **View Details...** You should see something like this:



The certificates and provisioning profiles for this account

Click the **Download** button to load the new **StoreSearch Ad Hoc** provisioning profile into Xcode.

Great, you're just about ready to build the app for distribution.

★ ★ ★

Debug builds vs. Release builds

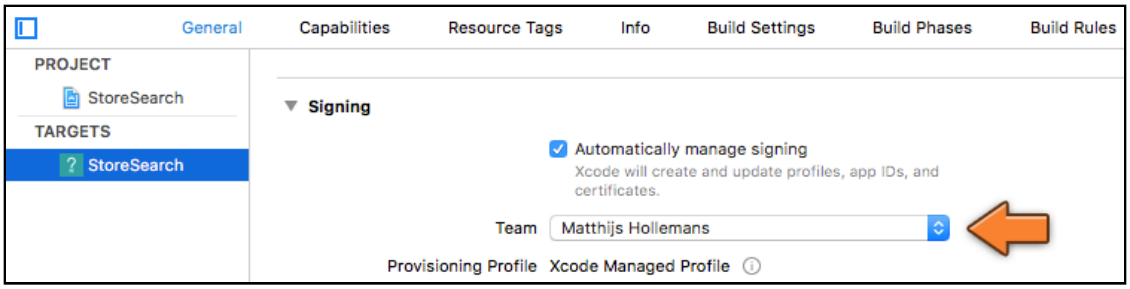
Xcode can build your app in a variety of **build configurations**. Projects come standard with two build configurations, Debug and Release. While you were developing the app you've always been using Debug mode, but when you build your app for distribution it will use Release mode.

The difference is that in Release mode certain optimizations will be turned on to make your code as fast as possible, and certain debugging tools will be turned off. Not including the debugging tools will make your code smaller and faster – they're not much use to an end user anyway.

However, changing how your app gets built does mean that your app may act differently under certain circumstances. Therefore it's a good idea to give your app a thorough testing in Release mode as well, preferably by doing an Ad Hoc install on your own devices. That is the closest you will get to simulating what a user sees when he downloads your app from the App Store.

You can add additional build configurations if you want. Some people add a new configuration for Ad Hoc and another for App Store that lets them tweak the build settings for the different types of distribution.

➤ In the **Project Settings** screen, in the **General** tab, choose the correct **Team**. This determines which certificate and provisioning profile Xcode will use.



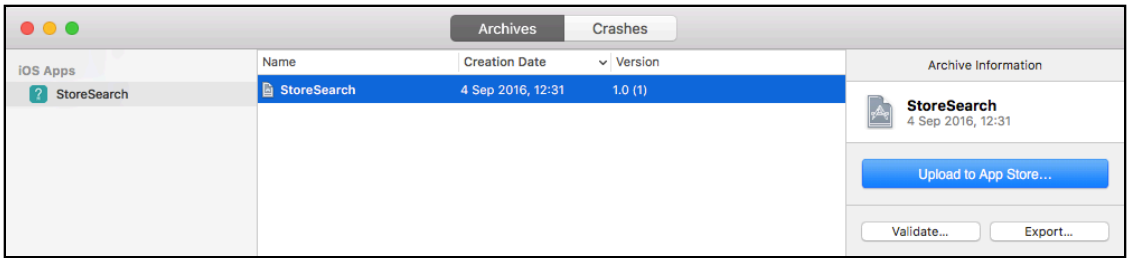
Choosing the team

➤ In the picker at the top of the Xcode window choose **Generic iOS Device** (or the name of your device if it is connected to your Mac) rather than a Simulator.

➤ From the Xcode menu bar, select **Product → Archive**. If the Archive option is grayed out, then the scheme is probably set to Simulator rather than the device.

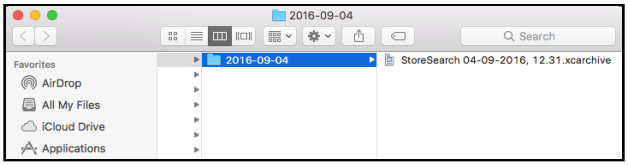
Now Xcode will build the app. By default, the Archive operation uses the Release build configuration.

➤ When the build is done and without errors, Xcode opens the Organizer window on the Archives tab:



The Archives section in the Organizer window

If you right-click the archive in the list and choose **Show in Finder**, the folder that contains the archive file opens:



The archive in Finder

By right-clicking the **.xcarchive** file and choosing **Show Package Contents**, you can take a peek inside. In the folder Products you will find the application bundle. To see what is in the application bundle, right-click it and choose Show Package Contents again.

★ ★ ★

dSYM files

The folder **dSYMs** inside the archive contains a very important file named **StoreSearch.app.dSYM**. This dSYM file contains symbolic names for the classes and methods in your app. That information has been stripped out from the final executable but is of vital importance if you receive a crash report from a customer. (You can see these crash reports in the Organizer window or download them through the iTunes Connect website.)

Crash reports contain heaps of numbers that are meaningless unless combined with the debug symbols from the dSYM file. When properly "symbolicated", the crash log will tell you where the crash happened – essential for debugging! – but in order for that to work Xcode must be able to find the dSYM files.

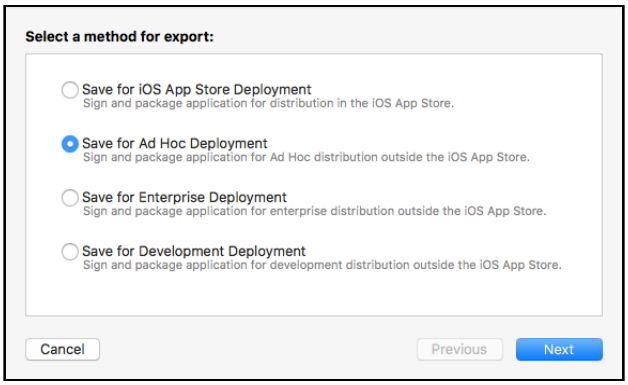
So it is important that you don't throw away these .xcarchive files for the versions of your app that you send to beta testers or the App Store. You don't have to keep them in the folder where Xcode puts them per se, but you should keep them around somewhere and back them up.

You don't want to get crash reports that you can't make any sense of! Even better, don't make apps that crash, so you won't get any crash reports at all…

★ ★ ★

The .xcarchive isn't the thing that you will send to your beta testers. Instead, Xcode will build another package that is based on the contents of this archive.

➤ In the Organizer window select the archive from the list and press the **Export...** button. In the screen that appears, select the **Save for Ad Hoc Deployment** option. Click **Next**.



Choosing the method of distribution

Now Xcode will ask for the team to use. Then it looks up the Ad Hoc provisioning profile and signs the app.

You may get a message that says **codesign wants to sign using key ... in your keychain**. This is Xcode asking for permission to use your distribution certificate. Click the **Always Allow** button or it will ask every time, which gets annoying quickly.

When it's done, Xcode puts a new folder on your Desktop with a **StoreSearch.ipa** file inside. This is the file that you will give to your beta testers. An IPA file is simply a ZIP file that contains a folder named "Payload" and your application bundle.

The .ipa file

Give this **.ipa** together with **StoreSearch_Ad_Hoc.mobileprovision** to your beta testers and they will be able to run the app on their devices.

This is what they have to do. It's probably a good idea for you to follow along with these steps, so you can verify that the Ad Hoc build actually worked.

1. Open iTunes and go to the Apps screen.

2. Drag **StoreSearch.ipa** into the Apps screen.
3. Drag **StoreSearch_Ad_Hoc.mobileprovision** file into the Apps screen.
4. Connect your iPhone or iPad to the computer.
5. Sync with iTunes.

That's it. Now the app should appear on the device. If iTunes balks and gives an error, then nine times out of ten you did not sign the app with the correct Ad Hoc profile or the user's device ID is not registered with the profile.

Ad Hoc distribution is pretty handy. You can send versions of the app to beta testers (or clients if you are into contract development) without having to upload the app to the App Store first.

There are practical limits to Ad Hoc distribution, primarily because it is intended as a testing mechanism, not as an alternative to the App Store. For example, Ad Hoc profiles are valid only for a few months, and you can only register 100 devices. You can reset these device IDs only once per year so be judicious about registering new devices.

It's a good idea to test your apps using Ad Hoc distribution before you submit them to the App Store, just so you're sure everything works as it's supposed to outside of Xcode.

TestFlight

iOS has a built-in beta testing service, TestFlight. In some ways this is simpler to use than Ad Hoc distribution, especially if you have many beta testers.

With TestFlight you no longer have to add the user's device ID (or UDID) to your development account. Instead you can send invitations to up to 1000 testers, per app. All a tester needs is an Apple ID and the TestFlight app.

Once they've accepted your invitation, the testers can install your beta version right from the TestFlight app. With this service your testers don't need to fuss with IPA files and iTunes anymore. It doesn't get much easier than that!

However, when you make your apps available through TestFlight they will be reviewed by Apple's App Store team first, something that can take a few days. And every update needs to be reviewed again. This is not needed with Ad Hoc builds.

Even if you use TestFlight for beta testing, it's still a good idea to make an Ad Hoc build for yourself before you submit the app to the store. This is your last chance to catch bugs for the app goes out to (paying) customers!

Read more on TestFlight: https://developer.apple.com/testflight/

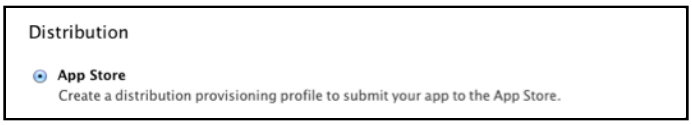
★ ★ ★

Submitting to the App Store

After months of slaving away at your new app, version 1.0 is finally ready. Now all that remains is submitting it to the App Store.

Doing so is actually fairly straightforward and I'll show you the steps here.

➤ You will need to create a new distribution profile on the iOS Member Center first. Go to **Provisioning Profiles**, **Distribution** in the sidebar and click the **+** button. This time you'll make an **App Store** profile.

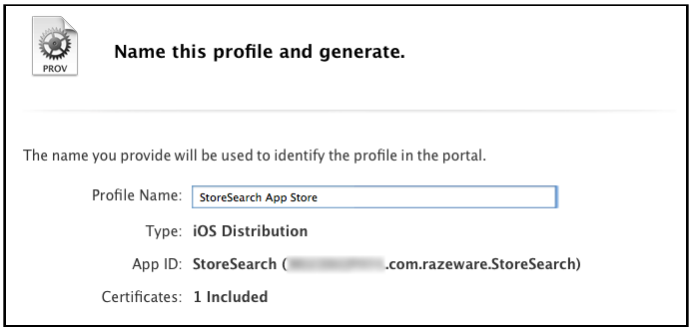


Choosing the App Store distribution profile

➤ The next step asks you for the App ID. Select the same App ID as before.

➤ The third step asks for your distribution certificate. Select the same certificate as before. There is no step for choosing devices; that is only required for Ad Hoc distribution.

➤ Give the profile the name **StoreSearch App Store** and click **Generate**.



Naming the provisioning profile

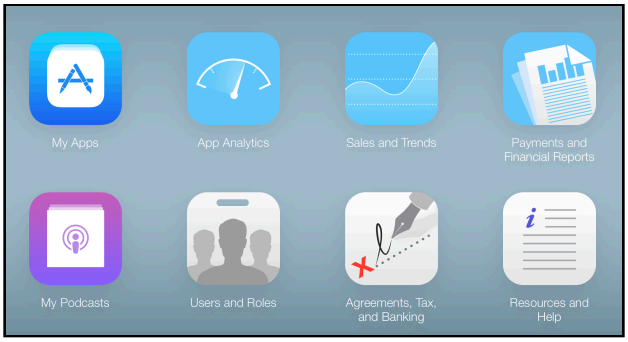
You don't have to download this provisioning profile, as Xcode will automatically fetch it from the Member Center when the time comes to sign the app.

You also do not have to re-build the app. You can use the archived version that you made earlier (no doubt you have tested the Ad Hoc version and found no bugs).

However, you first have to set up the application on iTunes Connect.

➤ Surf to itunesconnect.apple.com. Sign in using your Developer Program account.

If you've never been to iTunes Connect before, then make sure to first visit the **Agreements, Tax, and Banking** section and fill out the forms. All that stuff has to be in order before your app can be distributed on the App Store.



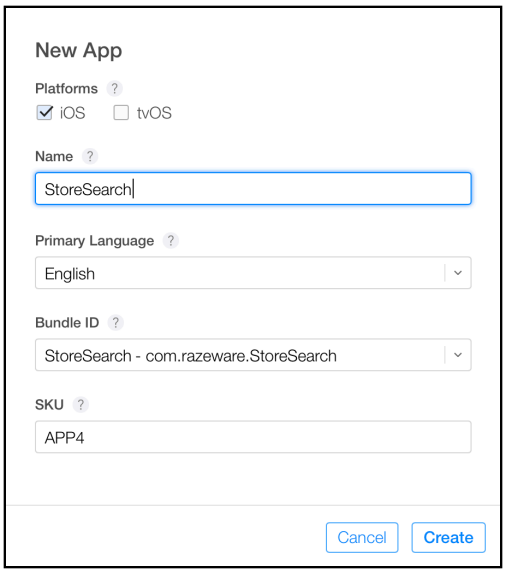
The iTunes Connect web site

> **Note:** The iTunes Connect interface changes from time to time, so what you see in your browser may be slightly different from these screenshots. The instructions that follow may not be 100% applicable anymore by the time you read this, but the general process of submitting an app will still be the same.

If this is the first app that you're adding, you will be asked to enter the name under which you wish to publish your apps on the store. You can use your own name or a company name, but choose carefully, you only get to pick this once and it's a big hassle to change later!

➤ When you've taken care of the administrativia, click on **My Apps** and then the **+** button and choose **New App**.

➤ Now is the time to enter some basic details about the app.



Entering the name and bundle ID of the app

I entered **StoreSearch** as the name for the app.

The SKU (or "skew") is an identifier for your own use; it stands for "stock-keeping unit". When you get sales reports, they include this SKU. It's purely something for your own administration.

For Bundle ID you pick the App ID that you used to make the distribution provisioning profile.

> **Note:** If your Bundle ID is not in the list, then make sure that it is not being used by one of your other apps (if you have them) and that you already made the distribution profile for it.

After you click **Create**, iTunes Connect presents you with the page that lets you enter the details for the new app. In the various sections you have to supply the following metadata about the app:

- The name of the app what will appear on the App Store
- The primary and secondary category that the app will be listed under
- You can upload up to five screenshots and one 30-second movie per device. You need to supply screenshots for 3.5-inch, 4-inch, 4.7-inch, and 5.5-inch iPhones, and the iPad. All these screenshots must be for Retina resolutions.
- A description that will be visible on the store
- A list of keywords that customers can search for (limited to 100 characters)
- A URL to your website and support pages, and an optional privacy policy

- A 1024×1024 icon image
- The version number
- Copyright information
- Your contact details. Apple will contact you at this address if there are any problems with your submission.
- A rating if your app contains potentially offensive material
- Notes for the reviewer. These are optional but a good idea if your app requires a login of some kind. The reviewer will need to be able to login to your app or service in order to test it.
- When your app should become available
- The price for the app
- Any In-App Purchases that you're offering

If your app supports multiple languages, then you can also supply a translated description, screenshots and even application name.

For more info, consult the iTunes Connect Developer Guide, available under Resources and Help on the home page.

Make a good first impression

People who are searching or browsing the store for cool new apps generally look at things in this order:

1. The name of the app. Does it sound interesting or like it does what they are looking for?
2. The icon. You need to have an attractive icon. If your icon sucks, your app probably does too. Or at least that's what people think and then they're gone.
3. The screenshots. You need to have good screenshots that are exciting and make it clear what your app is about. A lot of developers go further than just regular screenshots; they turn these images into small billboards for their app.
4. App preview video. Create a 15 to 30-second video that shows off the best features of your app.
5. If you didn't lose the potential customer in the previous steps, they might finally read your description for more info.

6. The price. If you've convinced the customer they really can't live without your app, then the price usually doesn't matter that much anymore.

So get your visuals to do most of the selling for you. Even if you can't afford to hire a good graphic designer to do your app's user interface, at least invest in a good icon. It will make a world of difference in sales.

The **Build** section in the **1.0 Prepare for Submission** section lists the actual app upload. Right now this section is empty.

Let's go back to Xcode so we can upload this guy!

➤ In the Xcode Organizer, go to the **Archives** tab, select the build you did earlier and choose **Validate**.

There are a bunch of things that can go wrong when you submit your app to the store (for example, forgetting to update your version number when you do an update, or a code signing error) and the Validate option lets you check this from within Xcode, so it's worth doing.

If you get an error at this point, double check that:

- The Bundle Identifier in Xcode corresponds with the App ID from the Dev Center and the Bundle ID that you chose in iTunes Connect.
- You have a valid iOS Distribution Certificate and an active App Store Distribution Profile for this App ID (check the iOS Dev Center).
- The **Team** is set up properly in the Xcode Project Settings screen.

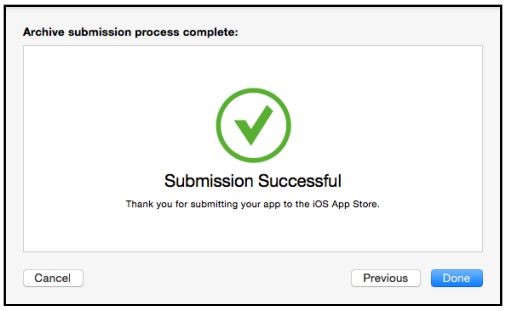
After fixing any of these issues, do **Product → Archive** again and validate the new archive.

Excellent! Now that the app checks out, you can finally submit it. This doesn't guarantee Apple won't reject your app from the store, it just means that it will pass the initial round of validations.

> **Note:** You don't have to submit your source code to Apple, only the final application bundle.

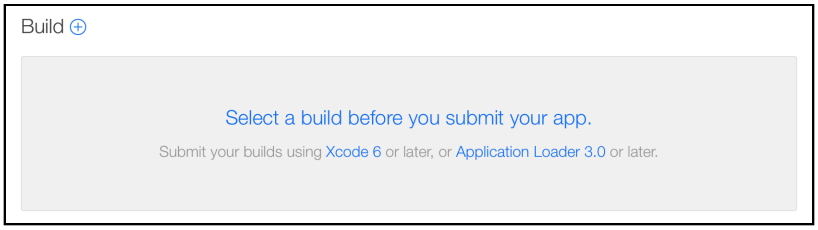
➤ In the Xcode Organizer, select the archive again and click **Upload to App Store**.

After a minute or two, you should see a confirmation:



And now the long wait begins…

Head back to iTunes Connect, reload the page for your app, and go to the Build section. There is a **+** button that lets you add a build.



Adding a build

This associates the archive you just uploaded with this app.

After filling out all the fields, click the **Save** button at the top. When you're ready to submit the app, press **Submit for Review**.

Your app will now enter the App Store approval process. If you're lucky the app will go through in a few days, if you're unlucky it can take several weeks. These days the wait time is fairly short. See http://appreviewtimes.com for an indication of how long you'll have to wait.

If you find a major bug in the mean time, you can reject the file you uploaded on iTunes Connect and upload a new one, but this will put you back at square one and you'll have to wait a week again.

If after your app gets approved you want to upload a new version of your app, the steps are largely the same. You go to iTunes Connect and create a new version for the app, fill in some questions, and upload the new binary from Xcode.

Updates take about the same amount of time to get reviewed as new apps, so you'll always have to be patient for a few days. (Tip: Don't forget to update the version number!)